UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matej Martinc

Učinkovito procesiranje naravnega jezika s Pythonom

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2015

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matej Martinc

Učinkovito procesiranje naravnega jezika s Pythonom

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Matjaž Kukar

Ljubljana, 2015

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Python je v zadnjih letih postal vodilni programski jezik in platforma za podatkovno analitiko, kamor sodi tudi procesiranje naravnega jezika (Natural Language Procession - NLP), tako splošne, kot v povezavi s spletno analitiko. Pojavile so se številne knjižnice, kot so npr. prevladujoča, široko usmerjena NLTK, ter alternative pattern, pyNLPl, spaCy in druge. Primerjajte in ovrednotite izbrane knjižnice za praktično uporabo na realističnem problemu. Uporabite različne kriterije, kot so funkcionalnost, hitrost, podpora različnim jezikom, integracija z jezikovnimi viri, ter ocenite njihovo primernost za delo s teksti v slovenskem jeziku.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Matej Martinc sem avtor diplomskega dela z naslovom:

Učinkovito procesiranje naravnega jezika s Pythonom

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matjaža Kukarja,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 31. julija 2015 Podpis avtorja:

Kazalo

Povzetek

Abstract

Poglavje 1 Uvod 1

Poglavje 2 Pregled knjižnic za procesiranje naravnega jezika v Pythonu 3

2.1 Natural language Toolkit (NLTK) 3

2.2 Knjižnica Pattern 4

2.3 Knjižnica SpaCy 5

2.4 Knjižnica PyNLPl 6

2.5 Knjižnica TextBlob 7

2.6 Primerjava knjižnic po funkcionalnosti 8

Poglavje 3 Pregled funkcionalnosti knjižnic za procesiranje naravnega jezika 11

3.1 Izdelava stavčnih dreves 11

3.2 Izdelava n-gramov 16

3.3 Dostop do korpusov besedil 18

3.4 Funkcije za iskanje vzorcev v besedilu 20

3.5 Štetje frekvenc besed 23

3.6 Podpora knjižnic procesiranju slovenskega jezika 25

Poglavje 4 Primerjava knjižnic po hitrosti in točnosti 28

4.1 Metodologija primerjave knjižnic po točnosti in hitrosti 28

4.2 Tokenizacija 29

4.2.1 Primerjava hitrosti tokenizacije 32

4.3 Lematizacija in korenjenje 34

4.3.1 Lematizacija 34

4.3.2 Korenjenje 34

4.3.3 Primerjava knjižnic: lematizacija ... 35
4.3.4 Primerjava hitrosti korenjenja ... 42
4.4 Oblikoskladenjsko označevanje ... 43
4.4.1 Opis označevalnikov ... 43
4.4.2 Primerjava označevalnikov po točnosti ... 47
4.4.3 Primerjava oblikoskladenjskih označevalnikov po hitrosti ... 50
4.5 Primerjava hitrosti kombinacije operacij ... 53
Poglavje 5 Klasifikacija in analiza sentimenta ... 55
5.1 Izbira teksta ... 56
5.2 Predprocesiranje teksta ... 57
5.3 Izračun značilk ... 58
5.4 Klasifikatorji ... 59
5.4.1 Naivni Bayesov klasifikator ... 60
5.4.2 Odločitvena drevesa ... 60
5.4.3 Klasificiranje z maksimalno entropijo ... 61
5.4.4 Metoda podpornih vektorjev (SVM) ... 61
5.4.5 Metoda najbližjih sosedov (KNN) ... 61
5.4.6 Enonivojski perceptron (SLP) ... 61
5.4.7 Uporaba klasifikacije v knjižnici NLTK ... 61
5.5 Kriteriji uspešnosti klasifikacije in primerjave klasifikatorjev ... 62
5.5.1 Meritve ... 64
5.6 Meritve hitrosti učenja klasifikatorjev ... 65
5.7 Izboljšave uspešnosti klasifikacije ... 67
5.7.1 Filtriranje s pomočjo oblikoskladenjskega označevanja ... 67
5.7.2 Filtriranje s pomočjo TF-IDF ... 70
5.7.3 Drugi poskusi izboljšave uspešnosti klasifikacije ... 75
5.8 Interoperabilnost knjižnic ... 78
5.9 Avtomatska izdelava učne množice tvitov za določanje sentimenta ... 79
5.10 Leksikalni pristop ... 81

5.10.1 Metoda merjenja in rezultati meritev 81
Poglavje 6 Sklepne ugotovitve 84
6.1 Ocene knjižnic 84
6.2 Izboljšave 86
6.3 Priporočila 87

Seznam uporabljenih kratic

kratica	angleško	slovensko
NLTK	Natural language toolkit	Knjižnica za obdelavo naravnega jezika
KNN	K – nearest neighbours	K – najbližjih sosedov
SVM	Support vector machine	Metoda podpornih vektorjev
API	Application program interface	Programski vmesnik
XML	Extensible Markup Language	Razširljiv označevalni jezik
TF-IDF	Term frequency – inverse document frequency	Mera za določanje pomembnosti besed v množici dokumentov
URL	Uniform resource locator	Spletni naslov
FIFO	First in first out	Algoritem za določanje vrstnega reda vhodov in izhodov
AUC	Area under curve	Mera za ocenjevanje uspešnosti klasifikacije
ARPA	Advanced research projects agency	Format za izdelavo jezikovnih modelov
RDR	Ripple down rules	Metoda za učenje pravil za lematizacijo
SLP	Single layer perceptron	Enonivojski perceptron
PMI	Pointwise mutual information	Mera asociacije
AFINN	Arup Finn Nielsen	Lista besed z njihovimi sentimentalnimi vrednostmi

Povzetek

V diplomski nalogi smo se v programskem jeziku Python lotili primerjave različnih orodij in knjižnic za procesiranje naravnega jezika. Poleg najbolj razširjene knjižnice NLTK smo raziskali še manj znane knjižnice, kot so SpaCy, PyNLPl, Pattern in TextBlob, ter jih med seboj primerjali na podlagi različnih kriterijev in praktičnih nalog, kot so tokenizacija, lematizacija, korenjenje, oblikoskladenjsko označevanje, izdelava stavčnih dreves, iskanje vzorcev v besedilu, štetje frekvenc besed v besedilu ter izdelava n-gramov. Knjižnice smo med seboj primerjali po kriteriju funkcionalnosti in v praksi preverili, katere metode in orodja za obdelavo naravnega jezika posamezne knjižnice vsebujejo, ter njihovo delovanje predstavili s praktičnimi primeri. Pozorni smo bili tudi na dostop do različnih korpusov besedil in na možnost procesiranja slovenskega jezika. Najpogostejše metode za obdelavo naravnega jezika, kot so tokenizacija, lematizacija in oblikoskladenjsko označevanje, smo med seboj primerjali na podlagi kriterijev hitrosti in točnosti. Nato smo se lotili določanja sentimenta slovenskih tvitov in komentarjev s pomočjo strojnega učenja ter pri tem preizkusili množico klasifikatorjev iz knjižnic NLTK, Pattern in TextBlob ter izmerili njihovo točnost in hitrost. Točnost klasifikacije smo poskusili izboljšati s pomočjo filtriranja na podlagi oblikoskladenjskega označevanja, filtriranja na podlagi TF-IDF, s pomočjo vključitve bigramov v učno množico ter s pomočjo odstranitve URL naslovov in ponavljajočih znakov. Na koncu smo metodo določanja sentimenta s pomočjo strojnega učenja primerjali še z leksikalno metodo. Zaključili smo z ugotovitvami, da je knjižnica PyNLPl zaradi odsotnosti orodij za lematizacijo, korenjenje in oblikoskladenjsko označevanje za procesiranje naravnega jezika neprimerna. Knjižnica NLTK je počasna, a zaradi obsežne dokumentacije in množice alternativnih metod primerna za učenje, odlikuje pa se tudi s fleksibilnostjo, zaradi katere je edina knjižnica, ki nudi delno podporo tudi procesiranju slovenskega jezika. Knjižnici SpaCy in TextBlob sta se izkazali s hitrostjo in točnostjo, a sta precej nefleksibilni in vsebujeta premalo funkcionalnosti. Knjižnica Pattern se je po kriterijih hitrosti in točnosti izkazala za povprečno, odlikuje pa jo predvsem fleksibilnost, množica funkcionalnosti in preprosta uporaba.

Ključne besede: procesiranje naravnega jezika, Python, lematizacija, tokenizacija, oblikoskladenjsko označevanje, analiza sentimenta.

Abstract

The thesis deals with a comparison of different tools and libraries for natural language processing in Python programming language. In addition to the most popular library for natural language processing NLTK we thoroughly researched other less known libraries, such as SpaCy, pyNLPl, Pattern and Textblob, and made comparisons between them based on different criteria and practical assignments, such as tokenization, lemmatization, stemming, part of speech tagging, dependency tree building, searching for patterns in text, word frequency counting and n-grams building. The libraries were compared by functionality and their methods and tools for natural language processing were analysed and described in detail. Special attention was paid to library abilities to access different text corpora and processing of Slovenian language. The most common methods of natural language processing, such as tokenization, lemmatization, part of speech tagging, were compared by speed and accuracy. Afterwards we focused on the sentiment analysis of Slovenian tweets and internet comments by machine learning, where we tested classifiers from different libraries and compared them by speed and accuracy. We tried to improve accuracy of classifiers with different methods, such as part of speech filtering, filtering based on TF-IDF, n-grams inclusion and removal of URL's and character repetitions in words. The machine learning approach to sentiment analysis was compared to lexical approach. We came to a conclusion that PyNLPl library lacks the basic methods for natural language processing. NLTK library is slow but suitable for learning because of the huge documentation and a big set of alternative methods. It is also very flexible, which makes it the only library that partially supports processing of Slovenian language. Spacy and TextBlob libraries are very fast and also accurate but they lack flexibility and functionality. Pattern library turned out to be average in terms of speed and accuracy but is pretty flexible, has many functionalities and is easy to use.

Keywords: natural language processing, Python, lemmatization, tokenization, part of speech tagging, sentiment analysis.

Poglavje 1 Uvod

Procesiranje naravnega jezika se uporablja povsod – v spletnih iskalnikih, pri avtomatičnem preverjanju črkovanja, na mobilnih telefonih, v računalniških igrah ter celo v nekaterih pomivalnih strojih. Obvladovanje interneta, ki je cilj raznih internetnih iskalnikov, bi bilo brez strojne obdelave različnih naravnih jezikov zaradi ogromne množice podatkov nemogoča naloga [24]. Vendar pa kljub vsem različnim metodam in pristopom stroji naravnega jezika še vedno ne razumejo dobro. Potrebno je obilo znanja, veščin in celo nekaj sreče, da na internetu dobimo odgovore na vprašanja naslednjega tipa: Katere turistične kraje med Ljubljano in Postojno naj obiščem z omejenim proračunom? Kako so spletni komentatorji komentirali politično dogajanje v Sloveniji v preteklem tednu? Kakšne so napovedi za rast železarske industrije v naslednjem letu? Če bi želeli, da bi računalnik avtomatično odgovarjal na zgornja vprašanja, ga moramo naučiti veliko zapletenih tehnik in veščin, kot so ekstrakcija informacij iz besedila, logično sklepanje ter obnavljanje besedil. Te veščine v končni fazi sodijo v področje umetne inteligence, katere dolgoročen cilj je zgraditi inteligentne stroje, ki bi med drugim morali razumeti tudi naravni jezik.

Dolga leta je cilj, da bi stroji razumeli naravni jezik, veljal za pretežkega, a se je z novejšim razvojem tehnologij za procesiranje naravnega jezika izkazal za realnejšega, robustne metode za analizo naravnega jezika pa so postale vse bolj razširjene. Seveda so te tehnologije še daleč od želenega razumevanja naravnega jezika, problemi, ki jih je treba pri tem prebroditi, pa so kar številni in kompleksni. Zaenkrat velja, da računalniki še niso sposobni razumevati naravnega jezika na podoben način in s takšno lahkoto kot ljudje. Velike probleme jim povzročajo notranja dvoumnost naravnega jezika, sarkazem ter stavčna logika, ki je velikokrat povezana in razumljiva le s pomočjo semantike. Zaradi zgoraj naštetih problemov je tako vsako besedilo potrebno spremeniti v obliko, ki je računalnikom bolj razumljiva.

Tako dandanes obstajajo mnoge tehnike za procesiranje naravnega jezika, ki so postale predpogoj za skoraj vsako nadaljnjo računalniško obdelavo besedil. Najbolj splošno uporabne so tokenizacija (razdelitev besedila na 'žetone', ponavadi na besede in ločila), lematizacija (pretvorba besede v njeno osnovno obliko) in korenjenje (iskanje korena besede) ter velikokrat tudi oblikoskladenjsko označevanje besed (določitev, za kakšno slovnično zvrst besede gre) in izdelava stavčnih dreves (iskanje odnosov med besedami v stavku). Poleg njih

obstajajo tudi druge malo manj splošno uporabne tehnike, kot so na primer izdelava n-gramov, iskanje osebnih imen in določanje frekvenc besed. Te tehnike pomagajo jezik spremeniti v računalniku lažje berljivo obliko in so predpogoj za dandanes zelo popularno tekstovno rudarjenje ter klasifikacijo tekstov, kamor sodi tudi analiza sentimenta (prepoznavanje čustvene zaznamovanosti teksta), ki je ključna za razumevanje pomena.

Python je programski jezik s koreninami v akademskih krogih in trenutno najbolj popularen jezik za obdelavo naravnega jezika. Veliko je k temu pripomogla njegova knjižica Natural language toolkit (NLTK) [8], ki je kmalu postala najširše uporabljena knjižnica za procesiranje naravnega jezika. NLTK je nastala v univerzitetnem okolju kot učni pripomoček za učenje obdelave naravnega jezika, a so jo zaradi širokih možnosti njene uporabe kmalu posvojili tudi v raznih poslovnih okoljih. Poleg te zelo obsežne in široko usmerjene knjižnice so se sčasoma pojavile tudi druge, manjše in bolj ozko usmerjene knjižnice za procesiranje naravnega jezika, kot so knjižnice Pattern [10], pyNLPl [12] , SpaCy [13], TextBlob [14], Gensim [7] in MontyLingua [39].

Cilji te diplomske naloge so tako večplastni. Za začetek želimo preveriti, kakšne funkcionalnosti imajo posamezne knjižnice in kako se obnesejo pri zgoraj naštetih osnovnih metodah obdelave naravnega jezika (tokenizacija, lematizacija, korenjenje, oblikoskladenjsko označevanje), jih med seboj primerjati na podlagi različnih kriterijev, kot so hitrost in točnost, ter odkriti, za katere naloge so posamezne knjižnice najprimernejše in za katere najmanj. Oboroženi z rezultati teh eksperimentov se bomo nato lotili naslednjega cilja diplomske naloge, ki je praktična uporaba teh knjižnic pri analizi sentimenta v slovenskih elektronskih besedilih, kot so tviti in komentarji. Med seboj bomo primerjali določanje sentimenta s pomočjo strojnega učenja in leksikalne metode, prav tako bomo nekaj časa posvetili poskusu izdelave avtomatske učne množice.

Veliko od teh knjižnic je narejenih posebej za delo v angleščini, ki je najbolj razširjen jezik na internetu, zato bomo za začetek knjižnice med seboj primerjali s pomočjo angleških tekstov in korpusov. Seveda pa nas bo zanimalo tudi danes zelo aktualno področje strojne obdelave slovenskega jezika, zato bomo preverili, če je te knjižnice mogoče prilagoditi za rabo v slovenščini, in v primeru, da to ne bo možno, poskušali najti orodja, ki to omogočajo.

Poglavje 2 Pregled knjižnic za procesiranje naravnega jezika v Pythonu

Za začetek smo se lotili iskanja knjižnic za procesiranje naravnega jezika. Poleg najbolj znane in najobširnejše knjižnice NLTK smo našli še štiri knjižnice, ki med drugim nudijo tudi metode za obdelavo naravnega jezika: Pattern, SpaCy, PyNLPl in TextBlob. Poleg teh knjižnic smo na začetku želeli preveriti tudi knjižnici Gensim in MontyLingua, ki ju Wikipedia [28] priporoča kot knjižnici za procesiranje naravnega jezika v Pythonu. Na žalost nam knjižnico MontyLinguia ni uspelo naložiti na računalnik, saj njena domača spletna stran ne deluje, prav tako knjižnica ni dosegljiva prek Pythonovega razdeljevalca paketov (*package manager*) Pip. Knjižnico Gensim smo po premisleku izključili iz primerjav, saj ne ponuja modulov ali funkcij niti za tako osnovne operacije procesiranja naravnega jezika, kot je tokenizacija, ampak ponuja le zelo napredne in specifične module (na primer latentna semantična obdelava) za obdelavo ogromne količine dokumentov. Tako nam je na koncu ostalo le zgoraj omenjenih pet knjižnic. Uporabljali smo NLTK verzijo 3.0.4, TextBlob verzijo 0.9.1, SpaCy verzijo 0.84, PyNLPl verzijo 0.7.6.6 in Pattern verzijo 2.6. Gre za najnovejše verzije knjižnic, ki so bile dosegljive v času začetka izdelave diplomske naloge. Zaradi problemov s kompatibilnostjo knjižnice Pattern s Pythonom 3.2+, smo uporabljali Python 2.7. Funkcionalnosti posameznih knjižnic bomo na kratko opisali v nadaljevanju.

2.1 Natural language Toolkit (NLTK)

NLTK [8] je najbolj znana in najobširnejša odprtokodna knjižnica za obdelavo naravnega jezika v Pythonu. Poleg množice različnih metod za obdelavo naravnega jezika je njena največja prednost, da kot edina knjižnica ponuja možnost dostopa do ogromne zbirke korpusov, ki jo sestavljajo besedila v različnih jezikih in različnih žanrov. Tako v tej zbirki najdemo v več jezikov prevedena znana literarna dela, deklaracijo človekovih pravic ter novejša besedila z interneta, kot so različni znanstveni in poljudni članki, ocene filmov s spletnega portala IMDB ter drugo [4]. Odlikuje se tudi po fleksibilnosti, zaradi katere kot edina od obravnavanih knjižnic nudi delno podporo tudi za procesiranje slovenskega jezika.

NLTK vsebuje najobsežnejšo zbirko orodij in metod za obdelavo naravnega jezika, ki so dobro komentirana in preprosta za uporabo, vsebuje pa tudi module za dostop do raznih

drugih programskih knjižnic, kot je na primer knjižnica Wordnet [16]. Tako nudi mnogo orodij za tokenizacijo, več različnih algoritmov za lematizacijo in korenjenje v različnih naravnih jezikih ter zbirko algoritmov za oblikoskladenjsko označevanje. NLTK poleg teh metod vsebuje tudi orodja za klasifikacijo ter orodja za izdelavo stavčnih dreves. Vsebuje razna analitična orodja, kot so funkcije za štetje besed, izris grafov in histogramov ter orodja za ocenjevanje algoritmov. Knjižnica vsebuje tudi druge kompleksne metode za razumevanje naravnega jezika, kot so na primer orodja za izdelavo na značilkah temelječe slovnice (*feature based grammar*) ali pa orodja za predstavitev naravnega jezika s pomočjo logike prvega reda. Ta orodja presegajo obseg te diplome, zato se v njih ne bomo poglobili.

Medtem ko je obsežnost in možnost alternativ za vsako operacijo največja prednost knjižnice NLTK, pa se mnogi pritožujejo nad njeno okornostjo ter počasnostjo. NLTK je bila na začetku razvita v raziskovalne in učne namene in kot taka ni bila namenjena uporabi v industriji, kjer je optimizacija delovanja prioriteta. Tako lahko na internetu preberemo mnogo komentarjev, da sta jasnost delovanja in dobra dokumentacija prevladala nad hitrostjo in točnostjo, kar nameravamo v tej diplomi preveriti v praksi.

2.2 Knjižnica Pattern

Pattern [10] je odprtokodna Pythonova knjižnica, ki je nastala na raziskovalnem oddelku za računsko lingvistiko in psiholingvistiko na univerzi v Antwerpnu. Ta knjižnica vsebuje množico uporabnih modulov (spletni pajek, Twitter API, Facebook API, API za dostop do podatkovnih baz, API-ji za branje XML in drugih vrst datotek, moduli za strojno učenje, moduli za vizualizacijo...), med njimi pa najdemo tudi modul za obdelavo naravnega jezika, ki nudi podporo za procesiranje angleškega, francoskega, španskega, nemškega, italjanskega in nizozemskega jezika. Ta knjižnica ne nudi podpore za procesiranje slovenskega jezika.

Znotraj tega modula najdemo funkcije za najbolj osnovne operacije pri obdelavi naravnega jezika, kot so tokenizacija, stematizacija in lematizacija. Slednja temelji na knjižnici Wordnet, ki je pravzaprav korpus angleških besed, med seboj povezanih s pomočjo relacij sopomenk, nadpomenk in podpomenk. Korpus pa vsebuje tudi osnovne oblike vseh angleških besed - leme. Knjižnica Pattern ima tudi funkcijo za oblikoskladenjsko označevanje, ki temelji na leksikonu 100000 besed in pravilih za kategorizacijo neznanih besed na podlagi končnic in konteksta.

Knjižnica poleg teh osnovnih operacij za procesiranje naravnega jezika vsebuje še modul za analizo sentimenta na podlagi leksikona čustveno zaznamovanih besed ter različna orodja za

nadzorovano in nenadzorovano klasifikacijo. Naslednje funkcije so prav tako povezane z obdelavo naravnih jezikov:

- funkcija za izdelavo n-gramov,
- orodje za preverjanje črkovanja,
- funkcija za iskanje vzorcev v besedilu (iskanje na podlagi oblikoslovnih označb, regularnih izrazov ter na podlagi sintaktičnih vzorcev),
- razne statistične funkcije (štetje besed v besedilu, izračun TF-IDF v dokumentu, kosinusna razdalja med dokumenti),
- orodja za ocenjevanje uspešnosti klasifikacije (preciznost, priklic, konfuzijska matrika),
- orodja za izdelavo značilk.

Poleg teh dokaj običajnih funkcij ima knjižnica Pattern še nekaj funkcij, ki so unikatne in jih nismo našli v nobeni drugi knjižnici. Med te sodijo funkcije za singularizacijo in pluralizacijo samostalnikov, orodja za skladnjo glagolov ter funkcija za zaznavo modalnosti glagolov. Zanimiv je tudi genetski algoritem za izdelavo novih, a vendarle izmišljenih »besed« na podlagi izvornih besed.

2.3 Knjižnica SpaCy

SpaCy [13] je preprosta knjižnica za obdelavo naravnega jezika, ki stavi predvsem na hitrost. Knjižnica je prosto dostopna za raziskovalno ali ljubiteljsko uporabo, medtem ko je potrebno za komercialno uporabo kupiti licenco. Knjižnica SpaCy ne podpira operacijskega sistema Windows. Podpira samo angleščino (ne nudi podpore za procesiranje slovenskega jezika) in vsebuje cevovod (pipeline), ki izvorno besedilo tokenizira, nato pa besedam določi oblikoskladenjske označbe, lemo in zgradi tako imenovani drevo odvisnosti (dependency tree) za vsak stavek. Znotraj tega cevovoda se posameznim besedam tudi določi povprečna frekvenca (na podlagi že preštetega korpusa treh milijard angleških besed). Poleg tega cevovoda vsebuje še dodatne funkcije, ki so povezane s procesiranjem naravnega jezika:

- vektorsko reprezentacijo besed, ki omogoča lažjo primerjavo med besedili (na primer s pomočjo kosinusne razdalje),
- funkcijo za izločitev predpone in končnice iz besede,

- razne druge preproste funkcije, kot so funkcija, ki pove, ali je beseda naslov besedila, in funkcija, ki pove, ali beseda spominja na URL naslov, ter funkcija, ki pove, kakšno identifikacijo ima beseda znotraj Brownovega razvrščanja (Brown clustering – hierarhično grupiranje besed glede na kontekst, v katerem se pojavljajo, ki ga je predlagal Peter F. Brown) [36].

SpaCy je napisan v programskem jeziku Cython, ki poleg pisanja Pythonove kode omogoča tudi klicanje funkcij v programskem jeziku C, hkrati pa se vsa napisana koda pred izvajanjem prevede v programski jezik C, ki slovi po svoji hitrosti. Zaradi tega naj bi bila knjižnica SpaCy najhitrejša Pythonova knjižnica za procesiranje naravnega jezika, k hitrosti pa pripomore tudi cevovod, ki s smiselnim združevanjem posameznih operacij procesiranja naravnega jezika, kot so tokenizacija, lematizacija in oblikoskladenjsko označevanje, optimizira celoten proces obdelave. Njeno hitrost bomo v nadaljevanju preverili v praksi.

2.4 Knjižnica PyNLPl

PyNLPl [12] je odprtokodna knjižnica, ki za razliko od drugih knjižnic ne vsebuje vseh osnovnih modulov za obdelavo naravnega jezika, kot sta lematizacija in oblikoskladenjsko označevanje, zato ni primerna za obdelavo slovenskega jezika. Vsebuje le modul za tokenizacijo in nekaj zanimivih orodij za obdelavo naravnega jezika:

- funkcijo za izdelavo n-gramov,
- funkcijo za izračun prekrivanja različnih besedilnih sekvenc,
- funkcijo za iskanje ključnih besed glede na kontekst.

Poleg teh modulov ta knjižnica vsebuje še vrsto orodij, ki niso direktno povezana s procesiranjem naravnega jezika:

- podatkovne strukture za shranjevanje (FIFO vrsta, vrsta s prioriteto, trie),
- funkcije za evalvacijo (AUC krivulja, F-mera, matrika klasifikacij, preciznost, priklic, specifičnost),
- funkcijo za vzorčenje, ki naredi testno in učno množico iz datoteke.

Knjižnica PyNLPl vsebuje tudi orodja za branje različnih formatov datotek (folia, GIZA++, Moses, Sonar, Taggerdata, Timbl) in orodja za izdelavo preprostih jezikovnih modelov, ki temeljijo na modelu ARPA. PyNLPl-jev modul za statistiko vsebuje funkcije za izračun

mediane, standardnega odklona in normalizacije, funkcije za izdelavo seznama frekvenc in distribucije, funkcijo za izdelavo skritega Markovskega modela, Markovske verige in izračun Levenstheinove razdalje.

2.5 Knjižnica TextBlob

TextBlob [14] je preprosta odprtokodna knjižnica za procesiranje naravnega jezika, ki jo je razvil Steven Loria in je primerna predvsem za procesiranje angleškega jezika (ta knjižnica ne nudi podpore za procesiranje slovenskega jezika) Knjižnico odlikuje predvsem preprostost uporabe in dobra ter pregledna dokumentacija. Posebnost te knjižnice je njena eklektičnost, saj je veliko orodij za obdelavo naravnega jezika preprosto vzeto iz drugih knjižnic. Pogled v dokumentacijo nam tako pove, da knjižnica za oblikoskladenjsko označevanje uporablja označevalnik knjižnice Pattern (možno je uporabiti tudi privzeti označevalnik knjižnice NLTK), za klasifikacijo pa imamo na izbiro veliko algoritmov iz knjižnice NLTK. Vsi ti tuji algoritmi so dobro integrirani v samo knjižnico in dostopni prek razreda 'TextBlob', ki je nekakšen ovojni razred za besedilni niz ('string') z veliko metodami za procesiranje naravnega jezika. Ta knjižnica za razliko od nekaterih prejšnjih vsebuje samo metode, ki so povezane s procesiranjem naravnega jezika, in sicer lahko v njej najdemo sledeča orodja:

- orodja za oblikoskladenjsko označevanje,
- orodja za tokenizacijo,
- funkcijo za lematizacijo,
- orodja za določanje sentimenta v angleških besedilih,
- orodja za klasifikacijo.

Poleg teh dokaj standardnih orodij za procesiranje naravnega jezika najdemo tudi metode za iskanje osebnih imen v besedilu, metodo za določanje sentimenta v angleškem jeziku, dostop do knjižnice Wordnet, orodja za pluralizacijo in singularizacijo samostalnikov v angleškem jeziku, metodo za določanje frekvenc besed v besedilu, orodja za izdelavo n-gramov, orodja za razčlenjevanje besedila (tudi to orodje uporablja razčlenjevalnik knjižnice Pattern) ter orodje za preverjanje črkovanja. Knjižnica je načeloma narejena za procesiranje angleškega jezika, vendar pa ima tudi metodo za prevajanje in detekcijo jezika, ki uporablja Googlov prevajalnik.

2.6 Primerjava knjižnic po funkcionalnosti

Za boljši pregled nad funkcionalnostjo posameznih knjižnic si poglejmo tabelo (Tabela 2.1), v kateri smo poskušali zajeti vse funkcionalnosti knjižnic, ki so povezane z obdelavo naravnega jezika.

	NLTK	**Pattern**	**SpaCy**	**PyNLPl**	**TextBlob**	**OpenNLP (Java)**
Tokenizacija	✔	✔	✔	✔	✔	✔
Lematizacija	✔	✔	✔		✔	✔
Korenjenje	✔	✔				✔
Oblikoskladenjsko označevanje	✔	✔	✔		✔	✔
Klasifikacija	✔	✔			✔	✔
Izdelava stavčnih dreves	✔	✔	✔			✔
Izdelava n-gramov	✔	✔		✔	✔	✔
Dostop do korpusov besedil	✔					✔
Funkcije za iskanje vzorcev v besedilu	✔	✔	✔	✔	✔	✔
Štetje frekvenc besed	✔	✔	✔	✔	✔	
Delna podpora slovenščini	✔					✔

Tabela 2.1: Primerjava knjižnic po funkcionalnosti

Funkcionalnosti knjižnic smo prav tako primerjali z najbolj popularno knjižnico za procesiranje naravnega jezika v programskem jeziku Java, OpenNLP [9], saj smo želeli preveriti, če ta knjižnica iz drugega programskega jezika vsebuje drugačen nabor funkcionalnosti.

Kot vidimo iz zgornje tabele, obstajajo med knjižnicami precejšnje razlike glede funkcionalnosti in vsebovanja metod za obdelavo naravnega jezika. Za začetek lahko opazimo, da obstajajo le tri metode, ki so skupne vsem Pythonovim knjižnicam. Te so tokenizacija, različne funkcije za iskanje vzorcev v besedilu ter štetje frekvenc besed. Medtem ko je princip tokenizacije v vseh knjižnicah enak in enako velja tudi za štetje frekvenc, pa se funkcije za iskanje vzorcev v besedilu od knjižnice do knjižnice razlikujejo. Več o tem bomo povedali v nadaljevanju, kjer bomo te funkcije predstavili s primeri.

Zanimivo je, da knjižnica PyNLPl ne vsebuje metod za lematizacijo, korenjenje in metode za oblikoskladenjsko označevanje, ki so dokaj temeljne operacije znotraj področja procesiranja naravnega jezika. Po drugi strani ni presenetljivo, da le tri knjižnice vsebujejo metode za klasifikacijo teksta, saj gre pri klasifikaciji za operacijo, ki gradi na drugih bolj temeljnih operacijah in kot taka ni temeljna metoda za procesiranje naravnega jezika. Zanimivo je tudi dejstvo, da kar štiri Pythonove knjižnice vsebujejo funkcije za izdelavo n-gramov, med njimi tudi knjižnica PyNLPl, ki ne vsebuje po našem mnenju bolj temeljnih metod za obdelavo naravnega jezika, kot sta lematizacija in korenjenje.

Iz zgornje tabele je prav tako razvidno, da je knjižnica NLTK edina Pythonova knjižnica, ki omogoča dostop do velike zbirke korpusov besedil v različnih jezikih. Knjižnica NLTK je prav tako edina Pythonova knjižnica, za katero lahko pogojno rečemo, da vsebuje podporo tudi za procesiranje slovenskega jezika, zakaj je tako, pa bomo povedali v naslednjem podpoglavju, kjer bomo podrobneje in s pomočjo primerov opisali posamezne funkcionalnosti knjižnic.

Primerjava Pythonovih knjižnic z javansko knjižnico OpenNLP nam pokaže, da je nabor orodij in metod za procesiranje naravnega jezika v obeh programskih jezikih podoben. Knjižnica OpenNLP vsebuje podobna orodja (manjka le funkcija za štetje frekvenc) kot najpopularnejša knjižnica za procesiranje naravnega jezika v Pythonu, knjižnica NLTK, poleg teh pa vsebuje še nekatera druga orodja, kot je na primer orodje za detekcijo stavkov v besedilu, in metodo, ki zaznava imena v tekstu. Vendar pa nobena od teh dveh metod ni temeljna metoda za obdelavo naravnega jezika, ampak bolj posebnost te knjižnice. In kot smo videli v prejšnjem poglavju, imajo tudi Pythonove knjižnice določena zelo specifična orodja, ki jih v tej tabeli zaradi njihove zelo specifične uporabnosti sploh nismo omenili. Tako kot

NLTK je tudi knjižnica OpenNLP precej prilagodljiva, zato nudi določeno podporo tudi procesiranju v slovenščini.

Načeloma je mogoče v Pythonu za procesiranje naravnega jezika uporabiti tudi javanske knjižnice. Za javanski knjižnici openNLP in CoreNLP [33] so bili narejeni pythonovski vmesniki (gre za tako imenovane ovojne razrede, ki z javanskimi knjižnicami komunicirajo prek strežnika), ki omogočajo uporabo njunih metod in orodij v programskem jeziku Python. Druge javanske knjižnice za procesiranje naravnega jezika, kot je na primer knjižnica ClearNLP [34], pa je mogoče v Pythonu uporabljati s pomočjo raznih orodij (na primer orodje Py4J [11]), ki omogočajo Pythonovemu prevajalniku, da dostopa do javanskih objektov v Javinem navideznem stroju (*Java virtual macine*). Ker je v tej diplomi poudarek na proučevanju nativnih Pythonovih knjižnic za procesiranje naravnega jezika, javanskih knjižnic ne bomo temeljito proučili ali uporabljali, vendar pa je možnost njihove uporabe vsekakor potrebno omeniti.

Poglavje 3 Pregled funkcionalnosti knjižnic za procesiranje naravnega jezika

V tem poglavju bomo na kratko opisali posamezne funkcionalnosti knjižnic, ki so povezane s procesiranjem naravnega jezika, in s primeri kode pokazali, kako se določene metode za procesiranje naravnega jezika uporabljajo v posameznih knjižnicah. Izpustili bomo opis metod tokenizacije, lematizacije, korenjenja in oblikoskladenjskega označevanja, saj bomo te metode podrobneje opisali v naslednjem poglavju, kjer jih bomo med seboj primerjali po hitrosti in točnosti.

3.1 Izdelava stavčnih dreves

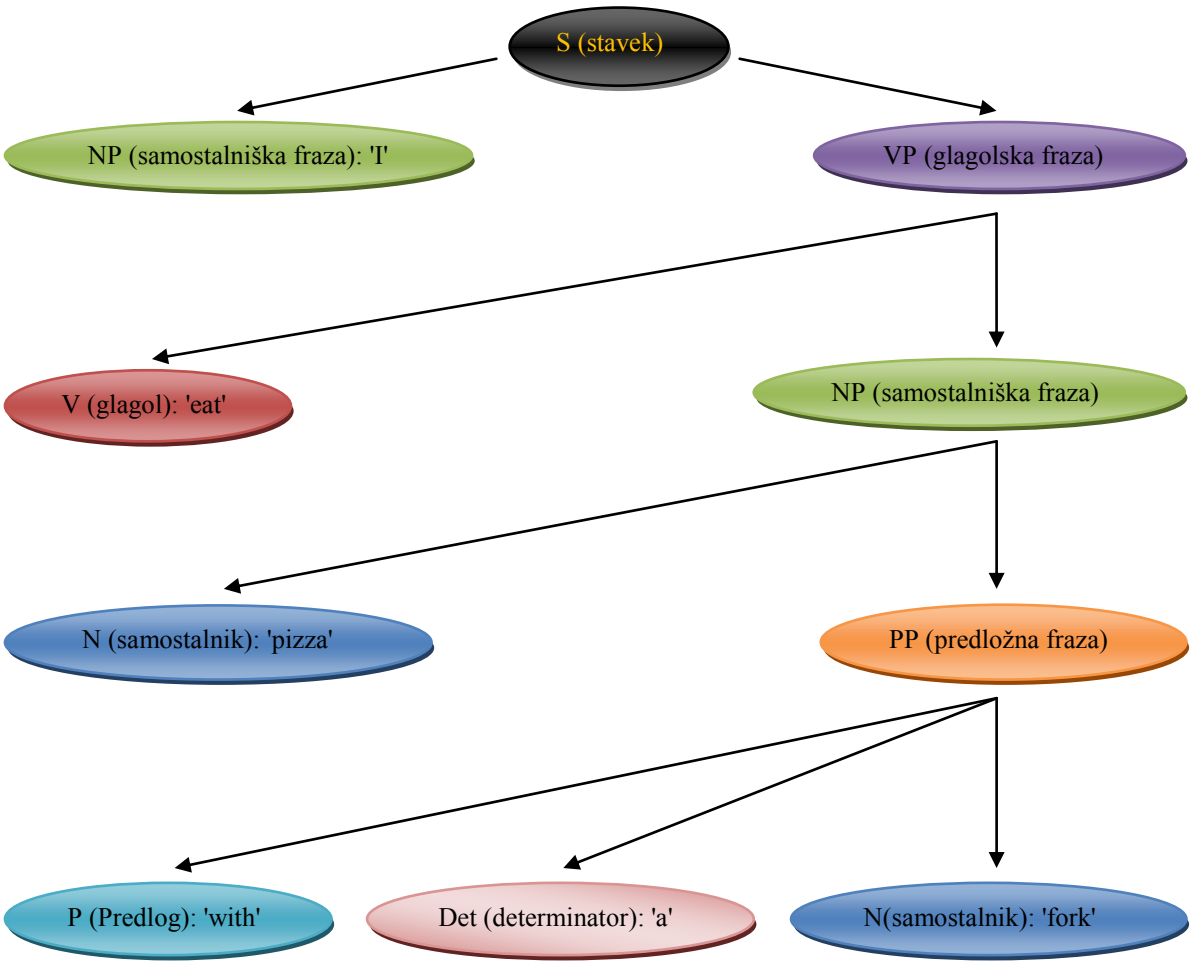
Knjižnice SpaCy, Pattern in NLTK vsebujejo module za izgradnjo stavčnih dreves, s pomočjo katerih lahko opišemo odnose med besedami v stavku. Za primer si poglejmo, kakšno stavčno drevo izdela knjižnica NLTK, če ji kot vhod damo angleški stavek »I eat pizza with a fork« (Jaz jem pico z vilico).

```
import nltk

#definiranje slovnice, ki določa strukturo drevesa
grammar = nltk.CFG.fromstring("""
  S -> NP VP
  VP -> V NP | V NP PP
  PP -> P NP
  V ->  "eat"
  NP -> "I" |Det N |Det N PP | N | N PP
  Det -> "a"
  N -> "pizza" | "fork"
  P -> "with"
  """)

#inicializacija rekurzivnega razčlenjevalnika, ki iz vhodnega stavka na
#podlagi definirane slovnice naredi vsa možna stavčna drevesa
parser = nltk.RecursiveDescentParser(grammar)

#razčlenjevalnik kot vhod zahteva že tokeniziran stavek v obliki seznama
sent = ['I', 'eat', 'pizza', 'with', 'a', 'fork']

#izpišimo vsa drevesa
for tree in parser.parse(sent):
    print tree
```

Izhod, ki ga dobimo, če poženemo zgornjo kodo, sta dve drevesi, ki ustrezata naši definirani slovnici (brez definirane slovnice razčlenjevalnik ne deluje). Da ne bomo preveč kompliciral, se bomo posvetili le enemu izmed njiju. Njegova predstavitev s pomočjo oklepajev je naslednja:

```
(S
  (NP I)
  (VP (V eat) (NP (N pizza) (PP (P with) (NP (Det a) (N fork))))))
```

Za boljšo predstavo si poglejmo vizualizacijo tega drevesa (Slika 3.1).



Slika 3.1: Grafični prikaz stavčnega drevesa, ki ga zgradi knjižnica NLTK

Vidimo, da gre pravzaprav za razčlenjevalno drevo, ki dele stavka deli na podlagi vnaprej definirane slovnice. Tako se v našem drevesu stavek najprej razdeli na samostalniško in glagolsko frazo, glagolska fraza se naprej razdeli na glagol in samostalniško frazo, ki se

naprej razdeli na samostalnik in predložno frazo in tako naprej. Na ta način dobimo stavek v drevesni obliki, ki predstavlja hierarhične odnose med besedami v stavku. Problem izdelave takih dreves je, da potrebujemo že vnaprej definirano slovnico za celotno besedilo, ki ga želimo predstaviti z drevesi, kar pri velikih besedilih predstavlja problem, saj nam ni uspelo nikjer najti že izdelane slovnice za angleška besedila, izdelava slovnice za celoten korpus, ki vsebuje več kot 100000 besed, pa je naloga, ki po svoji težavnosti in kompleksnosti presega namen te diplome. Po drugi strani nam knjižnica NLTK na ta način zagotavlja izredno fleksibilnost pri definiranju različnih odnosov med besedami, kar je v določenih primerih vsekakor prednost.

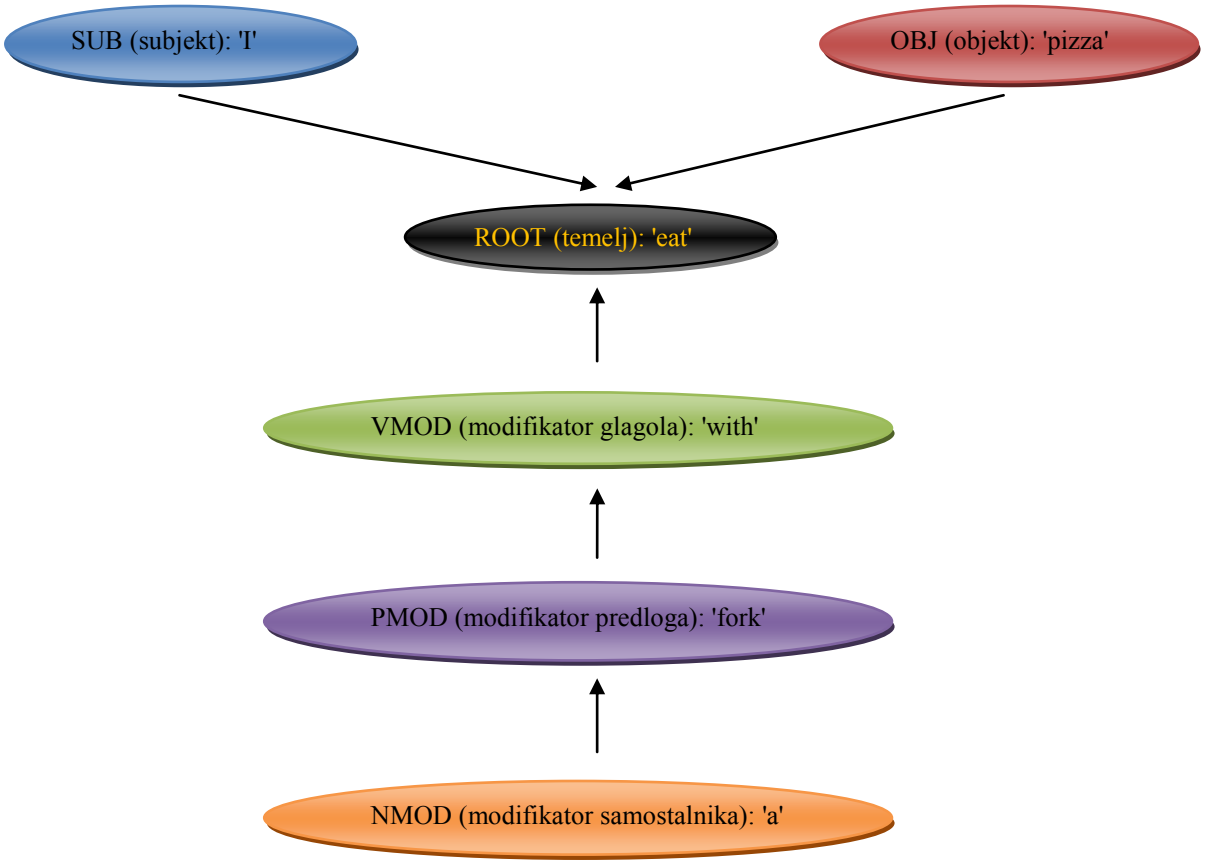
Poglejmo si, kakšna drevesa izdeluje knjižnica SpaCy:

```
from spacy.en import English

#inicializacija knjižnice
nlp = English()

#vhodni stavek
sentence = 'I eat pizza with a fork'

#vhodni stavek najprej zakodiramo v format 'unicode' in ga tokeniziramo
tokens = nlp(sentence.decode('unicode-escape'))

#spremenimo generator v seznam
listTokens = list(tokens)

#izpis drevesa v obliki seznama dvojic (beseda, odnos)
print [(word.orth_, word.dep_) for word in listTokens]
```

Zgornja koda nam vrne naslednji izpis:

[(u'I', u'SUB'), (u'eat', u'ROOT'), (u'pizza', u'OBJ'), (u'with', u'VMOD'), (u'a', u'NMOD'), (u'fork', u'PMOD')]

Poglejmo si še vizualizacijo tega drevesa (Slika 3.2):



Slika 3.2: Grafični prikaz stavčnega drevesa, ki ga zgradi knjižnica SpaCy

To drevo se razlikuje od drevesa, ki ga je naredila knjižnica NLTK. Gre za tako imenovano sintaktično drevo odvisnosti (*syntactic dependency tree*) [2]. V teh drevesih vsaka beseda modificira drugo besedo, ali pa je modificirana od druge besede (na sliki puščice kažejo smer modifikacije). Kot temelj je ponavadi postavljen glagol. To drevo nam dobro pokaže subjekt in objekt v stavku ter prikaže odnose med besedami, ki lahko pomagajo pri strojnem razumevanju pomena stavka. Hkrati opazimo, da se to drevo precej razlikuje od drevesa, ki ga je naredila knjižnica NLTK, saj gre pravzaprav za povsem drugo vrsto drevesa z drugačnimi slovničnimi označbami in odnosi.

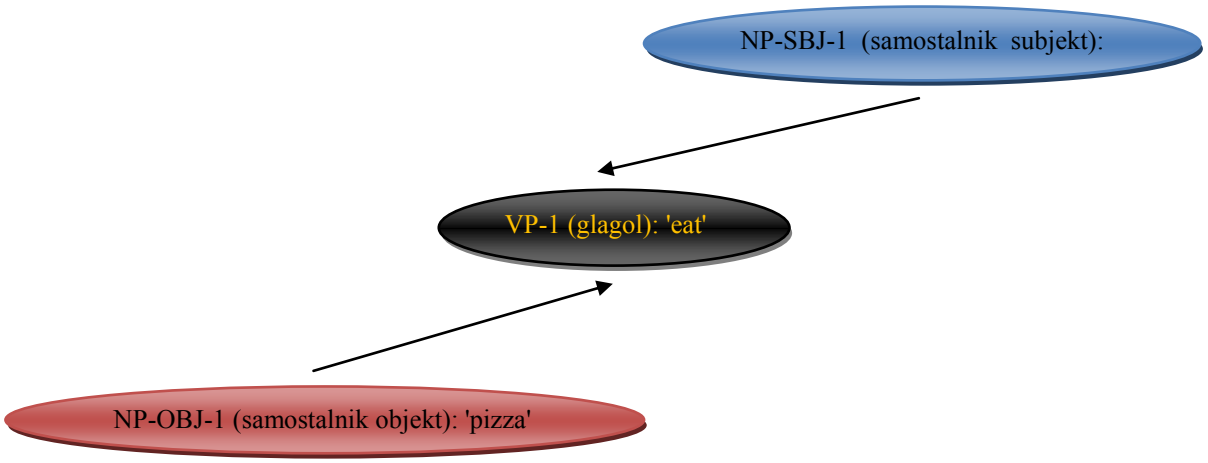
Poglejmo si še, kakšna drevesa dela knjižnica Pattern:

```
from pattern.text.en import parse

#za izdelavo drevesa poskrbi funkcija parse, s parametrom
#'relations' nastavljenim na 'True'
sentence = parse('I eat pizza with a fork', relations=True, tags=False,
chunks=False )

#uredimo izpis tako, da iz njega izločimo nerelevantne informacije
#ter besedilni niz spremenimo v seznam dvojic (beseda, odnos) za boljšo
#preglednost
print [(x.split("/")[0], x.split("/")[4]) for x in sentence.split(" ")]
```

Zgornja koda da naslednji izhod:

[(u'I', u'NP-SBJ-1'), (u'eat', u'VP-1'), (u'pizza', u'NP-OBJ-1'), (u'with', u'O'), (u'a', u'O'), (u'fork', u'O')]

Vizualizacija tega drevesa (Slika 3.3):



Slika 3.3: Grafični prikaz stavčnega drevesa, ki ga zgradi knjižnica Pattern

Vidimo, da gre za okrnjeno verzijo drevesa, ki ga je naredila knjižnica SpaCy, saj Patternov razčlenjevalnik v drevo vključi le subjekt, objekt in glagol. Znak 'O' je oznaka, ki se dodeli vsem ostalim besedam, ki niso vključene v drevo.

Drevesa, ki jih zgradijo knjižnice NLTK, SpaCy in Pattern, se med seboj razlikujejo. S pomočjo knjižnice NLTK smo zgradili razčlenjevalno drevo. Knjižnici SpaCy in Pattern po drugi strani zgradita sintaktično drevo odvisnosti (*syntactic dependency tree*), pri čemer je knjižnica SpaCy izdelala bolj podrobno drevo odvisnosti, ki vključuje vse besede v stavku.

Enaka drevesa kot v knjižnici Pattern je mogoče zgraditi tudi v knjižnici TextBlob, ki za gradnjo dreves uporablja kar Patternov razčlenjevalnik. Knjižnica PyNLPl gradnje dreves ne omogoča.

3.2 Izdelava n-gramov

N-grami so na splošno zaporedne, med seboj prekrivajoče se sekvence objektov dolžine n znotraj daljšega zaporedja objektov. V primeru procesiranja naravnega jezika so ti objekti običajno besede (žetoni znotraj tokeniziranega teksta), včasih pa tudi fonemi. N-grami se pri obdelavi naravnega jezika uporabljajo za določanje verjetnosti, da določeni besedi sledi druga beseda, in so kot taki uporabni pri prepoznavanju govora (*speech recognition*), pri izdelavi Markovskega modela ter pri različnih vrstah klasifikacije tekstov. Knjižnice NLTK, Pattern in PyNLPl vsebujejo module za izdelavo n-gramov, zato si oglejmo, kako je izdelava n-gramov videti v praksi v posameznih knjižnicah. Za začetek si poglejmo knjižnico NLTK:

```
from nltk import bigrams

#kot argument funkcija za izdelovanje bigramov potrebuje tokeniziran tekst
tokens = ["I", "eat", "pizza", "with", "a", "fork", "."]

#naredimo seznam bigramov
bigram_list = bigrams(tokens)
print list(bigram_list)
```

Zgornja koda vrne naslednji izhod v obliki seznama dvojic zaporednih besed:

[('I', 'eat'), ('eat', 'pizza'), ('pizza', 'with'), ('with', 'a'), ('a', 'fork'), ('fork', '.')]

Kot lahko opazimo, v knjižnici NLTK obstaja funkcija za izdelavo bigramov, torej n-gramov dolžine 2, ki so najpogostejši n-grami. Poleg te funkcije obstaja tudi funkcija za izdelavo trigramov:

```
from nltk import trigrams

#kot argument funkcija za izdelovanje trigramov potrebuje tokeniziran tekst
tokens = ["I", "eat", "pizza", "with", "a", "fork", "."]

#naredimo seznam trigramov
trigram_list = trigrams(tokens)
print list(trigram_list)
```

Tukaj je izhod seznam trojic zaporednih besed:

[('I', 'eat', 'pizza'), ('eat', 'pizza', 'with'), ('pizza', 'with', 'a'), ('with', 'a', 'fork'), ('a', 'fork', '.')]

Problem bi nastopil, če bi želeli večje n-grame, na primer n-grame dolžine štiri, saj knjižnica NLTK za to ne nudi podpore. Poglejmo si, kako izdelujemo n-grame v knjižnici Pattern:

```
from pattern.text.en import ngrams

#naredimo seznam bigramov
print ngrams("I eat pizza with a fork", n=2)
```

Zgornja koda vrne naslednji izhod:

[('I', 'eat'), ('eat', 'pizza'), ('pizza', 'with'), ('with', 'a'), ('a', 'fork')]

Opazimo lahko, da Patternova funkcija za izdelavo n-gramov za vhod ne potrebuje že tokeniziranega teksta, ampak tekst pred izdelavo n-gramov tokenizira sama. Prav tako pri tej funkciji ni omejitve glede velikosti n-gramov, saj s pomočjo argumenta 'n' določimo, kakšno dolžino n-gramov želimo.

Poglejmo si še knjižnico PyNLPl:

```
from pynlpl.textprocessors import Windower

#kot argument funkcija za izdelovanje n-gramov potrebuje tokeniziran tekst
tokens = ["I", "eat", "pizza", "with", "a", "fork", "."]

#izpišimo bigrame
print [bigram for bigram in Windower(tokens,2)]
```

Zgornja koda vrne naslednji izhod:

[(u'<begin>', 'I'), ('I', 'eat'), ('eat', 'pizza'), ('pizza', 'with'), ('with', 'a'), ('a', 'fork'), ('fork', '.'), ('.', u'<end>')]

Vidimo, da gre za podoben seznam, kot sta ga izpisali prejšnji dve knjižnici, razlika je le v oznakah '<begin>' in '<end>' na začetku in na koncu seznama. Opazimo lahko, da razred 'Windower', ki skrbi za izdelavo n-gramov, kot vhod zahteva že tokeniziran tekst, omogoča pa tudi, da s pomočjo argumenta definiramo velikost n-gramov. Poglejmo si še knjižnico TextBlob:

```
from textblob import TextBlob

#naredimo objekt TextBlob, ki vsebuje metodo za izdelavo n-gramov
blob = TextBlob("I eat pizza with a fork.")

#izpišimo bigrame
print blob.ngrams(n=2)
```

Izhod je sledeč seznam:

[WordList(['I', 'eat']), WordList(['eat', 'pizza']), WordList(['pizza', 'with']), WordList(['with', 'a']), WordList(['a', 'fork'])]

Kot vidimo, gre za seznam objektov 'WordList' (razred, ki deduje po Pythonovem seznamu in nudi nekaj dodatnih metod za obdelavo naravnega jezika). Sama funkcija ima enako ime kot Patternova funkcija za izdelavo n-gramov in tudi pri tej funkciji z argumentom 'n' določamo dolžino n-gramov.

3.3 Dostop do korpusov besedil

Knjižnica NLTK je edina knjižnica, ki omogoča dostop do velike zbirke korpusov besedil v različnih jezikih. Prek njenega programskega vmesnika je možno dostopati do 97 različnih korpusov, ti korpusi pa vsebujejo besedila v različnih jezikih, čeprav je največ tekstov v angleščini. Znotraj te ogromne zbirke besedil lahko najdemo deklaracijo o človekovih pravicah v 300 različnih jezikih ter znana literarna dela v več svetovnih jezikih. Zanimiva sta tudi korpus filmskih kritik z ročno določenim sentimentom (sentiment kritike je določen z oznako, ki pove, ali je kritika pozitivna ali negativna) in korpus angleških osebnih imen.

Dostop do teh korpusov je preprost: korpus preprosto uvozimo s pomočjo ukaza 'import':

```
from nltk.corpus import treebank
```

S tem ukazom smo uvozili korpus Treebank [18], ki ga bomo uporabljali v nadaljevanju. Posamezni korpusi imajo prav tako različne metode. Z metodo 'raw' dostopamo do neobdelanega teksta, z metodo 'words' pa do že tokeniziranega korpusa:

```
Treebank_string = treevank.raw()
treebank_words = treebank.words()
```

Vsi korpusi v knjižnici NLTK imajo te osnovne metode, obstajajo pa tudi korpusi z dodatnimi metodami. Naš korpus Treebank na primer vsebuje naslednji metodi:

```
tagged_sents = treebank.tagged_sents()
tagged_words = treebank.tagged_words()
```

Zgornji dve metodi nam vrneta seznam stavkov, v katerih so besedam pripisane oblikoskladenjske označbe ter seznam besed (korpus ni razdeljen na stavke ampak le na besede) s pripisanimi oblikoskladenjskimi označbami. Te metode nam bodo koristile v naslednjih poglavjih.

Pri knjižnici NLTK lahko uvažamo tudi lastne korpuse s pomočjo razreda 'PlaintextCorpusReader':

```
from nltk.corpus import PlaintextCorpusReader

#definirajmo pot do direktorija z besedili, ki jih želimo v korpusu
corpus_root = '/usr/share/diploma'

#ustvarimo korpus. Prvi argument je pot do korpusa, drugi je regularni
#izraz, ki določa, katere datoteke v direktoriju bodo vključene v korpus
wordlists = PlaintextCorpusReader(corpus_root, '.*')

#Izpišimo vse datoteke v korpusu
print wordlists.fileids()
```

Zgornja koda v našem primeru vrne izhod '['besedilo1', 'besedilo2', 'besedilo3']', kjer so naša besedila pravzaprav le tri prazne tekstovne datoteke v našem direktoriju '/usr/share/diploma', ki smo ga uporabili za ponazoritev delovanja uvoza korpusov v knjižnico NLTK.

Ostale knjižnice ne vsebujejo programskega vmesnika za dostop do že izdelanih korpusov besedil. Funkcije v teh knjižnicah kot argument sprejemajo le besedilo v obliki besedilnega niza (*string*), izjema je le Patternova funkcija 'parse', ki kot argument sprejema tudi tokeniziran tekst v obliki seznama besed. To načeloma ni velik problem, saj so v Pythonu lahko besedilni nizi zelo veliki, zato lahko tudi funkcijam v knjižnicah Pattern, SpaCy, TextBlob in PyNLPl kot argument dajemo ogromne korpuse besedil.

3.4 Funkcije za iskanje vzorcev v besedilu

Že zgoraj smo omenili, da vse knjižnice vsebujejo metode za iskanje vzorcev v besedilu. Te funkcije pa se med seboj precej razlikujejo. V tem poglavju bomo prikazali, katere funkcije posamezne knjižnice vsebujejo ter s primeri predstavili nekaj najzanimivejših.

Začnimo s knjižnico NLTK, ki vsebuje zanimivi funkciji »concordance« in »similar«:

```
import nltk

#tokeniziran tekst
tokens = ["I", "eat", "pizza", "with", "a", "fork", ".", "I", "eat",
"soup", "with", "a", "spoon", "."]

#NLTK-jev razred 'Text' nudi metode za iskanje po tekstu, kot argument pa
sprejme tokeniziran tekst
text = nltk.Text(tokens)

#klic funkcij za iskanje
print text.concordance("eat")
print text.similar("pizza")
```

Funkcija 'concordance' kot argument dobi besedo in kot izhod vrne vse pojavitve (oziroma privzeto do največ 25 pojavitev) te besede v kontekstu (privzeto je kontekst 75 znakov levo in 75 znakov desno od besed). Izhod v našem primeru je videti tako:

Displaying 2 of 2 matches:
I eat pizza with a fork . I eat soup with a
with a fork . I eat soup with a spoon .

Funkcija je našla dve pojavitve besede »eat« in jih vrnila skupaj s kontekstom. Poglejmo si še izhod za funkcijo »similar«:

soup

Ta funkcija kot argument dobi besedo in kot izhod vrne besede, ki se pojavljajo v podobnem kontekstu. V našem primeru je to beseda »soup«, saj vidimo, da je v našem tokeniziranem seznamu ta beseda obkrožena z istimi besedami (»I« in »with«) kot beseda »pizza«, ki smo jo dali funkciji kot argument.

Knjižnica PyNLPl vsebuje funkcijo »find_keyword_in_context«, ki deluje zelo podobno kot NLTK-jeva funkcija »concordance«. Edina razlika je, da kot argument zahteva tudi tokeniziran tekst, v katerem besedo iščemo, namesto da bi imela poseben razred z metodami

za iskanje. Ker je funkcija tako podobna funkciji »concordance«, je ne bomo opisali s primerom.

Nekaj zanimivih in uporabnih funkcij za iskanje ima tudi knjižnica Pattern:

```
from pattern.text.search import search

#iskanje in izpis
print search('pizza', 'I eat pizza with a fork')
```

Funkcija »search« v zgornjem primeru vrne naslednji izhod:

[Match(words=[Word('pizza')])]

Gre za seznam vseh ujemanj (Patternov razred 'Match'), saj bi načeloma lahko v tekstu iskali tudi več besed. Takšna funkcija za iskanje je sicer precej uporabna, vendar pa ni nič neobičajnega. V naslednjem primeru bomo videli, da je Patternova funkcija »search« sposobna tudi drugih, bolj kompleksnih iskanj:

```
from pattern.text.search import search, taxonomy
from pattern.text.en import parsetree

#izdelava taksonomije
for f in ('apple', 'pizza', 'yogurt'):
    taxonomy.append(f, type='food')

#izdelava stavčnega drevesa, ki ga bomo kot argument dali funkciji search
t = parsetree('I eat pizza with a fork.', lemmata=True)

#klic funkcije search, kot argument ji damo taksonomijo in stavčno drevo
print search('FOOD', t)
```

V zgornjem primeru smo najprej s pomočjo Patternove funkcije »taxonomy« izdelali taksonomijo hrane (različne vrste hrane smo shranili pod kategorijo »food« (hrana) ter izdelali stavčno drevo iz stavka »I eat pizza with a fork«. Oboje smo kot argument dali funkciji »search« in dobili naslednji izhod:

[Match(words=[Word(u'pizza/NN')])]

Funkcija je preiskala stavčno drevo za vse besede, ki spadajo pod kategorijo hrane in našla besedo »pizza«.

Knjižnica Pattern ima tudi funkcijo za iskanje slovničnih vzorcev v besedilu:

```
from pattern.text.en import parsetree

#izdelava stavčnega drevesa, ki ga bomo kot argument dali funkciji search
t = parsetree('I eat pizza with a fork.', lemmata=True)

#definiranje slovničnega vzorca, ki ga iščemo v tekstu. Iščemo
#samostalniške fraze, ki ju loči besedna zveza 'with a'
p = Pattern.fromstring('* {NP} with a {NP}')
m = p.match(t)

#izpis samostalniških fraz
print m.group(1)
print m.group(2)
```

V zgornjem primeru dobimo naslednja izhoda, ki ustrezata našemu iskanju:

[Word(u'pizza/NN')]
[Word(u'fork/NN')]

Izhod je torej seznam objektov 'Word', ki vsebujejo iskano frazo.

Tudi knjižnica TextBlob vsebuje zanimivo metodo za iskanje vzorcev v besedilu, in sicer gre za orodje za iskanje osebnih imen:

```
from textblob import TextBlob

#naredimo objekt TextBlob, ki vsebuje metodo za iskanje osebnih imen
blob = TextBlob("I eat Giovanni pizza with a fork.")

#izpišimo osebna imena
print blob.noun_phrases
```

Izhod zgornje kode je seznam osebnih imen, v našem primeru pa je videti tako:

['giovanni']

SpaCy-jeve funkcije za iskanje vzorcev v besedilu so bolj omejene in manj zanimive, saj knjižnica vsebuje le funkcije za identifikacijo naslovov in URL-jev ter funkcijo, ki pove identifikacijo besede znotraj Brownovega razvrščanja, zato se nam ne zdi potrebno, da bi jih predstavili s primerom.

Opazimo lahko, da se funkcije za iskanje vzorcev v besedilu v različnih knjižnicah med seboj razlikujejo. Največ funkcij za iskanje vzorcev vsebujeta knjižnici NLTK in Pattern, v knjižnicah SpaVy, TextBlob in PyNLPl pa je razpoložljivih funkcij za iskanje vzorcev manj.

3.5 Štetje frekvenc besed

Vse knjižnice vsebujejo funkcije za štetje frekvenc besed v besedilu, ki ga obdelujemo. V knjižnici NLTK za to poskrbi razred »FreqDist«:

```
from nltk import FreqDist

#tokeniziran seznam besed
tokens = ["I", "eat", "a", "pizza", "with", "a", "fork", "."]
print FreqDist(tokens).items()
```

Zgornja koda vrne naslednji izhod:

[('a', 2), ('fork', 1), ('I', 1), ('.', 1), ('with', 1), ('eat', 1), ('pizza', 1)]

Kot vidimo, razred »FreqDist« izdela slovar z besedami kot ključi in frekvencami kot vrednostmi. Knjižnica Pattern za izpis frekvenc posameznih besed uporablja funkcijo »count«, ki poleg samega izpisa frekvenc nudi še par dodatnih funkcionalnosti:

```
from pattern.vector import count

#tokeniziran seznam besed
tokens = ["I", "eat", "a", "pizza", "with", "a", "fork", "."]

#vključimo nerelevantne besede('stopwords')
print count(tokens, stopwords=True)

#izključimo nerelevantne besede
print count(tokens, stopwords=False)

#izpišimo le besede, ki se pojavijo več kot enkrat
print count(tokens, threshold=1, stopwords=True)
```

Zgornja koda vrne naslednje izhode:

{'a': 2, 'fork': 1, 'i': 1, '.': 1, 'with': 1, 'eat': 1, 'pizza': 1}
{'fork': 1, '.': 1, 'eat': 1, 'pizza': 1}
{'a': 2}

Izhodi so trije slovarji, kjer prvi vsebuje vse besede, drugi le besede, ki so relevantne (izključili smo vse tako imenovane 'stopwords'), tretji slovar pa vsebuje le besede, ki so se pojavile več kot enkrat.

Tudi knjižnica PyNLPl vsebuje razred za izračun frekvenc besed, in sicer razred 'FrequencyList', v katerem so besede in njihove frekvence spravljene v obliki slovarja na enak način kot v knjižnici NLTK.

```
from pynlpl.statistics import FrequencyList

freqlist =  FrequencyList()

#v slovar dodajmo tokeniziran seznam besed
freqlist.append(["I", "eat", "a", "pizza", "with", "a", "fork", "."])
print freqlist["a"]
```

Zgornja koda kot izhod vrne frekvenco besede »a«, torej 2.

Poglejmo si še knjižnico SpaCy:

```
from spacy.en import English, attrs

#inicializacija cevovoda
nlp = English()

#cevovodu kot argument damo 'unicode' besedilo
tokens = nlp(u'I eat a pizza with a fork', tag="False")

#funkcija za štetje frekvenc besed
print tokens.count_by(attrs.ORTH)

#funkcija, ki vrne logaritemsko normalizirane verjetnosti pojavitve besed,
#ki so določene vnaprej na podlagi preštetega korpusa
print [(word.orth_, word.prob) for word in tokens]
```

Knjižnica SpaCy, tako kot večino drugih operacij, izvede štetje besed znotraj svojega cevovoda. Kot izhod funkcija 'count_by' vrne naslednji slovar:

{36L: 1, 11L: 2, 14703L: 1, 57L: 1, 669L: 1, 23135L: 1}

Zanimivo je, da namesto besed v seznamu dobimo le reference na besede (identifikatorje). Domnevamo, da pripis identifikatorjev besedam pripomore k hitrejšemu delovanju cevovoda. Knjižnica SpaCy oziroma njen cevovod poleg zgornje funkcije za štetje vsebuje še atribut 'prob', ki vrne z logaritmom normalizirano verjetnost, da se posamezna beseda pojavi v besedilu. Ta verjetnost pa ni izračunana na podlagi frekvence besed v besedilu, ki ga obdelujemo, ampak na podlagi frekvence besed v že preštetem korpusu s tremi milijardami besed. Kot izhod zadnje vrstice kode tako dobimo naslednji seznam dvojic (beseda, log(p)):

[(u'I', -5.826796531677246), (u'eat', -10.452000617980957), (u'a', -4.003841400146484), (u'pizza', -12.134479522705078), (u'with', -5.231648921966553), (u'a', -4.003841400146484), (u'fork', -12.839383125305176)]

Tudi knjižnica TextBlob vsebuje metodo za določitev frekvenc besedam v besedilu:

```
from textblob import TextBlob

#naredimo objekt TextBlob, ki vsebuje metodo za izračun frekvenc
blob = TextBlob("I eat a pizza with a fork.")

#izpišimo seznam dvojic (beseda, frekvenca)
print blob.word_counts.items()
```

Izhod je sledeč:

[(u'a', 2), (u'fork', 1), (u'i', 1), (u'with', 1), (u'eat', 1), (u'pizza', 1)]

Vidimo, da tudi funkcija 'word_counts' naredi slovar, kjer so ključi besede, vrednosti pa frekvence.

Kot vidimo, prav vse knjižnice vsebujejo metodo za štetje frekvenc besed, kar pa ni presenetljivo, saj gre za eno od osnovnih statističnih metod, ki služi kot izhodišče za izračun drugih statističnih mer, kot je na primer mera TF-IDF, ki jo bomo uporabili tudi v tej diplomski nalogi.

3.6 Podpora knjižnic procesiranju slovenskega jezika

V tem poglavju bomo podrobneje obrazložili, zakaj je knjižnica NLTK edina obravnavana knjižnica, ki nudi delno podporo tudi obdelavi slovenskega jezika. V vseh petih knjižnicah obstajajo operacije, ki jih je možno izvajati v različnih jezikih. Poglejmo si spodnjo tabelo (Tabela 3.1). Pri tokenizaciji besedilo razdelimo na žetone (besede in ostale znake, kot so ločila). Ta operacija ponavadi deli besedilo na podlagi presledkov, zato jo je možno uporabljati v vseh fonetičnih jezikih, kjer besede pišemo s črkami abecede in jih med seboj ločimo s presledki. Problem nastopi pri logografskih jezikih, kot je na primer kitajščina, kjer imamo simbole, ki namesto fonemov predstavljajo celotne besede, med njimi pa ni presledkov. Tu operacija tokenizacije ne bi delovala pravilno. Enako velja za funkcijo za izdelavo n-gramov ter funkcijo za določanje frekvenc besed v besedilu, ki za svoje delovanje potrebujeta le predhodno tokenizacijo.

	Jezikovno specifična operacija	**Jezikovno nespecifična operacija**
Tokenizacija		✔
Lematizacija	✔	
Korenjenje	✔	
Oblikoskladenjsko označevanje	✔	
Izdelava n-gramov		✔
Štetje frekvenc besed		✔
Izdelava stavčnih dreves	✔	

Tabela 3.1: Tabela jezikovno specifičnih in jezikovno nespecifičnih operacij

Po drugi strani so lematizacija, korenjenje, oblikoskladenjsko označevanje ter izdelava stavčnih dreves operacije, ki so specifične za vsak jezik posebej. Operaciji lematizacije (iskanje osnovne oblike besed) in korenjenja (iskanje korena besede) potrebujeta za svoje delovanje seznam osnovnih oblik besed ali pa vsaj množico pravil za pretvorbo besede v njeno osnovno obliko ali koren. Ta pravila in seznami pa se razlikujejo od jezika do jezika. Tako se na primer za korenjenje v angleškem jeziku najpogosteje uporablja Porterjev algoritem [37], ki pa je za vse druge jezike neuporaben. Podobno velja za operaciji oblikoskladenjskega označevanja (pripis slovničnih oznak besedam) in izdelave stavčnih dreves (določanje odnosov med besedami v stavku), kjer je potrebno za delovanje poznati slovnico jezika, ki se od jezika do jezika razlikuje.

Nobena od knjižnic ne vsebuje lematizatorja za slovenski jezik, prav tako ne vsebuje algoritmov za iskanje korenov besed v slovenskem jeziku. Knjižnica NLTK je edina knjižnica, ki omogoča oblikoskladenjsko označevanje v slovenskem jeziku, saj edina omogoča učenje oblikoskladenjskih označevalnikov na podlagi lastno izbranih korpusov (ki so lahko tudi slovenski). Ostale knjižnice tega ne omogočajo. V knjižnici SpaCy je označevalnik že naučen na angleškem korpusu, v knjižnici Pattern imamo označevalnike naučene na korpusih v šestih jezikih (angleščini, nemščini, italijanščini, francoščini, španščini in nizozemščini), med katerimi pa ni slovenskega jezika. Knjižnica TextBlob za označevanje uporablja Patternov označevalnik in torej prav tako ne omogoča oblikoskladenjskega označevanja v slovenščini. Tako za oblikoskladenjsko označevanje v knjižnici NLTK

potrebujemo le učni korpus v slovenščini, ki pa ga knjižnica NLTK žal ne vsebuje. Podobno velja za izdelavo stavčnih dreves, saj je NLTK edina knjižnica, kjer je možno definirati slovnico, ki jo bomo uporabili za izdelavo dreves, medtem ko je v drugih knjižnicah ta slovnica definirana že vnaprej.

Zaradi te fleksibilnosti knjižnice NLTK lahko zaključimo, da nudi delno podporo za obdelavo slovenskega jezika, medtem ko je druge knjižnice ne nudijo, čeprav ta podpora ni zelo obsežna. Velik problem tako še vedno predstavlja lematizacija, ki jo s pomočjo teh knjižnic ni mogoče izvesti v slovenščini, vendar je predpogoj za skoraj vsako malo resnejše procesiranje naravnega jezika. Ta problem smo v nadaljevanju rešili s pomočjo odprto-kodne knjižnice Lemmagen [19], ki nudi podporo tudi za slovenski jezik.

Poglavje 4 Primerjava knjižnic po hitrosti in točnosti

4.1 Metodologija primerjave knjižnic po točnosti in hitrosti

Kot smo videli v prejšnjem poglavju, se vseh pet knjižnic po funkcionalnosti med seboj precej razlikuje. Vsaka poleg dokaj standardnih funkcij za obdelavo naravnega jezika vsebuje tudi zelo specifične funkcije za procesiranje naravnega jezika, ki jih druge knjižnice nimajo, in module, ki z obdelavo jezika sploh niso povezani.

Zaradi teh razlik med knjižnicami je primerjava različnih funkcionalnosti po kriterijih točnosti in hitrosti otežena. Že v prejšnjem poglavju smo omenili, da ima vseh pet knjižnic skupne le tri funkcionalnosti: tokenizacijo, dodelitev frekvenc besedam in razne funkcije za iskanje. Ker je tokenizacija ena od temeljnih metod vsakega procesiranja naravnega jezika, smo se odločili, da knjižnice med seboj primerjamo po hitrosti tokenizacije. Knjižnice bi lahko po hitrosti primerjali tudi pri operaciji izdelave n-gramov (to operacijo vsebujejo štiri knjižnice od petih) ter po operaciji dodelitve frekvenc besedam, vendar se nam te operacije niso zdele dovolj specifične in relevantne za področje procesiranja naravnega jezika, saj se ti dve operaciji uporabljata tudi zunaj področja obdelave naravnega jezika. Poleg tega gre za precej enostavni operaciji, ki jih je možno ob vsaj bežnem poznavanju programiranja izvesti brez pomoči knjižnic. Primerjavo raznih funkcij za iskanje po točnosti in hitrosti smo izključili zaradi prevelike raznolikosti in medsebojne različnosti teh funkcij. Iz istega razloga smo opustili tudi primerjavo hitrosti in točnosti knjižnic pri izdelavi stavčnih dreves, saj so drevesa, ki jih zgradijo posamezne knjižnice, med seboj preveč različna. Problem je predstavljala tudi izdelava slovnice za knjižnico NLTK, ki je preveč zahtevna naloga, če bi želeli knjižnice primerjati na malce večjih korpusih.

Po odločitvi za primerjanje knjižnic po hitrosti tokenizacije in izključitvi zgoraj naštetih funkcionalnosti so nam ostale le še bolj temeljne metode in operacije za obdelavo naravnega jezika, ki jih vsebuje večina knjižnic. Te operacije so lematizacija, korenjenje in oblikoskladenjsko označevanje ter so med seboj primerljive po hitrosti in točnosti. Knjižnica PyNLPl od vseh teh temeljnih metod vsebuje le modul za tokenizacijo, zato jo bo mogoče primerjati le po tem kriteriju.

Ker so vse knjižnice najbolj prilagojene za obdelavo angleškega jezika, določenih operacij (na primer oblikoskladenjskega označevanja) pa v knjižnicah Pattern in SpaCy (v teh dveh knjižnicah ni mogoče učiti oblikoskladenjskih označevalcev na lastni učni množici) ni mogoče izvesti v slovenskem jeziku, bomo knjižnice pri vseh teh štirih naštetih operacijah med seboj primerjali s pomočjo angleških korpusov, ki so dosegljivi prek vmesnika v knjižnici NLTK. Pri primerjavi hitrosti tokenizacije bomo zaradi njegove dolžine uporabili korpus Gutenberg [17], ki vsebuje 25000 elektronskih knjig in je nastal v okviru projekta Gutenberg. Pri primerjavi hitrosti in točnosti lematizacije, korenjenja in oblikoskladenjskega označevanja bomo uporabili korpus Penn Treebank [18] z nekaj več kot 100.000 besedami. Ta korpus vsebuje besede z ročno pripisanimi oblikoskladenjskimi označbami in ga zato lahko uporabimo kot zlati standard pri merjenju točnosti oblikoskladenjskih označevalnikov. Ker posamezne knjižnice vsebujejo cevovod, v katerem so smiselno združene vse zgoraj naštete operacije (to velja predvsem za knjižnico SpaCy, na cevovod pa spominja tudi Patternova funkcija 'parse'), bomo na koncu izmerili tudi hitrost kombinacije vseh operacij ter na ta način preverili, če implementacija cevovoda pospeši izvajanje skupka operacij. Hitrost kombinacije operacij bomo preverjali na korpusu Gutenberg.

4.2 Tokenizacija

Tokenizacija je razdelitev besedil v manjše enote (ponavadi besede in ločila), ki jim pravimo žetoni. Gre za osnovno operacijo, ki se uporablja kot prvi korak pri vsaki malce resnejši obdelavi besedila [24]. Vse knjižnice, ki jih primerjamo, vsebujejo tokenizacijo, vendar vse metode ne uporabljajo istega algoritma, zato tudi žetoni niso enaki.

Knjižnica NLTK vsebuje več funkcij za tokenizacijo, osnovna je funkcija 'word_tokenize', ki razdeli besedilo glede na presledke in ločila (ločil ne zavrže, ampak jih obravnava kot samostojne žetone). Ostale funkcije razdelijo besedilo samo po presledkih (razred 'WhitespaceTokenizer') ter drugače obravnavajo ločila (razred 'PunktWordtokenizer') in okrajšave. Osnovna 'word_tokenize' funkcija razdeli okrajšave (na primer »can't« se razdeli v »ca« in »n't«). Razred 'PunktWordTokenizer' razdeli besedilo po ločilih, vendar ločila niso samostojni žetoni, ampak so dodani besedi pred ločilom. Okrajšave ta razred razdeli po apostrofu. Če želimo večjo kontrolo nad produkcijo žetonov, potem je najbolje uporabiti razred 'RegexpTokenizer', kjer s pomočjo regularnih izrazov definiramo, kakšne žetone želimo, in sicer lahko izbiramo, ali želimo z regularnimi izrazi definirati žetone ali pa znake, ki besedilo razdelijo na žetone.

Knjižnica Pattern vsebuje tokenizator, ki razdeli besedilo po stavkih (gre za tako imenovani postopek segmentacije). Ločila obravnava ločeno in jih s presledkom loči od besede.

Tokenizator loči med okrajšavami in koncem stavka. Poglejmo si, kako deluje Patternov tokenizator, če vzamemo za primer besedilo »Danes je kras'n dan. Ali ste za to, da gremo na izlet?«.

```
from pattern.text.en import tokenize

#tokenizator knjižnice Pattern
print tokenize("Danes je kras'n dan. Ali ste za to, da gremo na izlet?")
```

Kot izhod dobimo naslednji seznam:

["Danes je kras'n dan .", 'Ali ste za to , da gremo na izlet ?']

Kot vidimo, je izhod seznam, ki vsebuje stavke, znotraj katerih so ločila, kot sta pika in vprašaj, od besed ločena s presledkom, medtem ko so okrajšane besede ('kras'n') obravnavane kot enovit žeton. Takšna oblika je primerna za nadaljnjo obravnavo, saj v primeru, da želimo namesto stavkov kot žetone uporabiti besede, vsak stavek razdelimo po presledkih in na ta način dobimo enak rezultat, kot ga za izhod vrne privzeti tokenizator v NLTK, torej funkcija 'word_tokenize'. Takšno operacijo izvede tudi Patternova funkcija parse, ki jo kličemo v primeru, ko želimo dobiti iz besedila še več informacij (na primer dodelitev oblikoskladenjskih označb posameznim besedam). Ta funkcija vrne po presledkih razdeljene stavke kot sezname, ki so med seboj ločeni z znaki za novo vrstico.

Knjižnica SpaCy vsebuje tokenizator, ki vrne besedilo, razdeljeno na besede. Tudi ta tokenizator loči med okrajšavami in okrajšavami v besedah ter ločila obravnava kot ločene žetone. Poglejmo si, kaj se zgodi z istim stavkom kot zgoraj, če ga tokeniziramo s tokenizatorjem knjižnice SpaCy:

```
from spacy.en import English

#tokenizator knjižnice SpaCy
def spacyTokenizer(s):

    #inicializacija knjižnice
    nlp = English()

    #tokenizacija s pomočjo SpaCy-jevega cevovoda in pretvorba generatorja
    #v seznam
    tokens = nlp(s.decode('unicode-escape'), tag=False)
    listTokens = list(tokens)

    #izpis žetonov in filtriranje drugih informacij, ki jih proizvede
    #cevovod
    return [word.orth_ for word in listTokens]

#klic funkcije
print spacyTokenizer("Danes je kras'n dan. Ali ste za to, da gremo na
izlet?")
```

Kot izhod dobimo naslednji seznam:

[u'Danes', u'je', u"kras'n", u'dan', u'.', u'Ali', u'ste', u'za', u'to', u',', u'da', u'gremo', u'na', u'izlet', u'?']

SpaCyjev tokenizator kot vhod ne sprejema besedilnega niza (*string*), ampak besedilo, kodirano kot 'unicode' besedilo, zato je treba vsak niz predhodno zakodirati v format 'unicode'. SpaCyjev tokenizator ne deluje kot samostojna funkcija, ampak je dosegljiv le v sklopu cevovoda, ki poleg žetonov vrne še oblikoskladenjske označbe, leme in druge informacije, ki jih včasih ne potrebujemo. S pomočjo opcijskega argumenta 'tag=False' lahko izključimo oblikoskladenjsko označevanje in izdelavo stavčnih dreves, kar močno pospeši proces tokenizacije, ne moremo pa izključiti lematizatorja ter nekaterih drugih časovno nezahtevnih funkcij, ki vrnejo informacije, ki jih ne potrebujemo.

Poglejmo si še PyNLPl-jev tokenizator:

```
import pynlpl.textprocessors as textprocessors

#tokenizator knjižnice pyNLPl
print textprocessors.crude_tokenizer("Danes je kras'n dan. Ali ste za to,
    da gremo na izlet?")
```

Zgornja koda vrne kot izhod naslednji seznam:

['Danes', 'je', "kras'n", 'dan', '.', 'Ali', 'ste', 'za', 'to', ',', 'da', 'gremo', 'na', 'izlet', '?']

Kot vidimo, tudi ta tokenizator razdeli besedilo na besede in ločila dojema kot posamezne žetone. Čeprav v dokumentaciji piše [12], da tokenizator ne upošteva okrajšav, pa lahko na zgornjem primeru vidimo, da je pravilno dojel besedo »kras'n« kot en žeton.

Nazadnje si poglejmo še tokenizator knjižnice TextBlob:

```
from textblob.tokenizers import word_tokenize as tbtokenizer

#tokenizator knjižnice TextBlob vrača generator, ki ga spremenimo v seznam
print list(tbTokenizer("Danes je kras'n dan. Ali ste za to, da gremo na
    izlet?"))
```

Rezultat zgornje kode je enak seznam besed kot pri knjižnici PyNLPl:

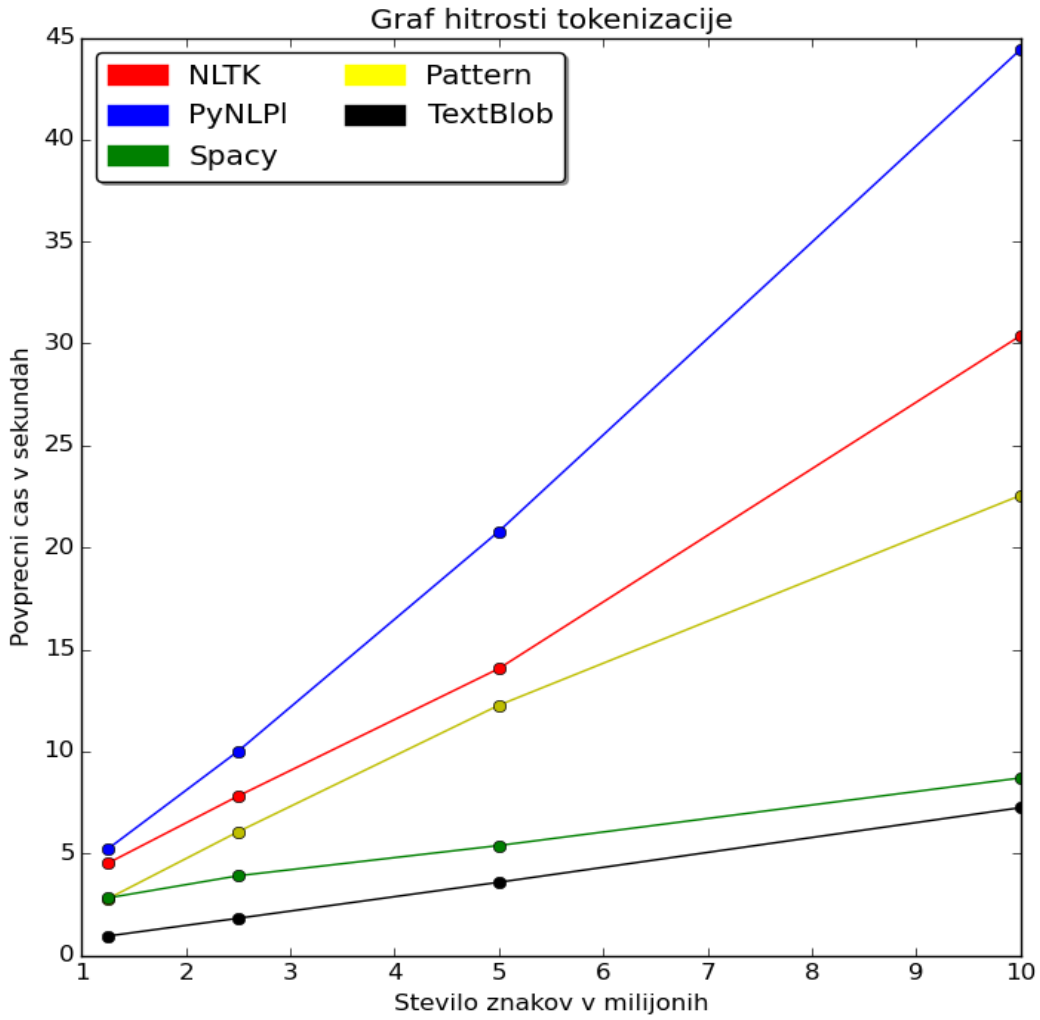
['Danes', 'je', "kras'n", 'dan', '.', 'Ali', 'ste', 'za', 'to', ',', 'da', 'gremo', 'na', 'izlet', '?']

Ločila so tudi tu obravnavana kot posamezni žetoni, okrajšave pa so upoštevane.

4.2.1 Primerjava hitrosti tokenizacije

Primerjava knjižnic po hitrosti tokenizacije da zanimive rezultate. Hitrost knjižnic smo testirali na NLTK korpusu Gutenberg. Sam korpus je dolg nekaj več kot 11 milijonov znakov, mi pa smo se odločili, da knjižnice testiramo na prvih 1250000 znakih, prvih 2500000 znakih, prvih 5 milijonih znakov ter na prvih 10 milijonih znakov tega korpusa. Hitrost smo merili s pomočjo Pythonove knjižnice Timeit [35], in sicer smo meritev tokenizacije v vsaki knjižnici izvedli desetkrat, v grafu pa je predstavljeno povprečje teh meritev. Ker ima knjižnica Pattern tokenizator, ki razdeli besedilo na stavke in le doda presledke med ločili, smo temu tokenizatorju dodali še operacijo razdelitve stavkov po presledkih (funkcija »split«), da smo na ta način dobili enak rezultat kot pri vseh ostalih knjižnicah, ter izmerili hitrost vseh teh operacij skupaj. Funkcija za merjenje hitrosti tokenizacije je zaradi svoje dolžine priložena v dodatku te diplomske naloge.

Slika 4.1 prikazuje hitrost tokenizatorjev. Kot vidimo, je najhitrejša knjižnica TextBlob, malce počasnejša pa je knjižnica SpaCy, ki sta edini uspeli opraviti z največjim korpusom v manj kot 10 sekundah. Razlika med njima je majhna, prav tako pa velja omeniti, da je standardni odklon pri meritvah za knjižnico SpaCy 0.2 sekundi, standardni odklon meritev pri knjižnici TextBlog pa 0.15 sekunde. Tretja najhitrejša, a skoraj trikrat počasnejša od prvih dveh knjižnic, je knjižnica Pattern. Za skoraj tretjino več časa od nje je pri najdaljšem korpusu porabil privzeti tokenizator NLTK. Najpočasnejši je tokenizator knjižnice pyNLPl, ki je za najdaljši korpus z 10 milijoni znakov porabil okoli 45 sekund, kar je več kot šestkrat počasneje od knjižnice TextBlob.



Slika 4.1: Primerjava knjižnic po hitrosti tokenizacije

Kot vidimo na grafu, čas tokenizacije glede na število znakov narašča linearno, iz grafa pa je razvidno tudi to, da je pri tekstih do 1250000 znakov vseeno, če uporabimo tokenizator knjižnice Pattern ali pa tokenizator knjižnice SpaCy, saj sta pri teh tekstih približno enako hitra. Kakorkoli, tudi za manjša besedila je priporočljiva knjižnica TextBlob, ki je s tekstom 1250000 znakov opravila več kot dvakrat hitreje od knjižnic Pattern in SpaCy. Ta knjižnica je tako pri vsaki tokenizaciji, ne glede na velikost korpusa, najboljša izbira.

4.3 Lematizacija in korenjenje

4.3.1 Lematizacija

Lematizacija je proces transformacije besede v njeno osnovno obliko. To je možno pri tistih besednih vrstah, ki so pregibne in tvorijo oblikoslovno paradigmo [21]. Pri lematizaciji za razliko od korenjenja velja, da je končni produkt vedno slovnično pravilna beseda. Ta operacija je ključna pri skoraj vsaki obdelavi naravnega jezika, kjer je cilj razumevanje tega jezika, saj nam omogoča enačenje vseh pomensko enakih besed, čeprav se zaradi svoje pregibnosti v tekstu pojavljajo v različnih časih, sklonih in spregatvah.

Za lematizacijo ponavadi potrebujemo tudi slovar besed v osnovni obliki, saj je nemogoče vse besede pretvoriti v osnovno obliko le s pomočjo raznih ročno določenih pravil, ker v naravnem jeziku obstaja množica izjem za vsako pravilo. Pri lematizaciji v angleškem jeziku nam zato prav pride korpus knjižnice Wordnet [16]. Wordnet je slovar angleških besed, narejen posebej za obdelavo naravnega jezika. Vsebuje množice angleških besed z istim pomenom (sopomenk) in je kot tak optimiziran za iskanje sopomenk za posamezne besede. Te množice sopomenk so med seboj povezane z različnimi hierarhičnimi odnosi (Wordnet ima strukturo hierarhičnega drevesa), kot so nadpomenskost in podpomenskost, zato lahko iščemo nadpomenke in podpomenke posameznim besedam, oziroma se sprehajamo po celotni hierarhični zgradbi angleškega jezika vse do korena (*root*), ki je v angleščini beseda entiteta (*entity*). Wordnet prav tako vsebuje vse možne leme za posamezno besedo.

Lema v Wordnetu lahko pripada le eni množici sopomenk, medtem ko beseda lahko pripada več množicam, saj je pomen besede velikokrat odvisen od konteksta (pri določanju konteksta nam ponavadi zelo pomaga oblikoskladenjska označba besede). Tako velja, da množica sopomenk predstavlja skupino lem, ki imajo enak pomen, medtem ko lema predstavlja osnovno obliko besede. Nekatere leme v Wordnetu imajo tudi protipomenke, prav tako lahko za določene leme poiščemo primere uporabe te besede v stavku ter definicijo besede. Zaradi vseh teh podatkov, ki jih Wordnet vsebuje, je ta slovar poleg drugih operacij zelo priročen tudi za izračun pomenske podobnosti med posameznimi množicami sopomenk, ki se izračuna tako, da pogledamo, koliko korakov je oddaljena skupna nadpomenka dveh množic sopomenk.

4.3.2 Korenjenje

Korenjenje ali stematizacija je tehnika za iskanje besednega korena. Tehnika se pogosto uporablja za indeksiranje besed v spletnih iskalnikih, ki namesto vseh oblik ene in iste besede

shranjujejo le korene [24]. Ta tehnika je zelo podobna tehniki lematizacije, le da tu velja, da je korenjenje možno izvesti le s pomočjo določenih pravil, ki se v različnih naravnih jezikih razlikujejo. Korenjenje je zaradi manjše kompleksnosti hitrejša operacija od operacije lematizacije, prav tako tu ne potrebujemo slovarja besed. Stematizacija se razlikuje od lematizacije po tem, da je rezultat lematizacije vedno slovnično pravilno beseda (osnovna oblika besede), medtem ko pri stematizaciji ostane besedni koren, ki ni nujno slovnično pravilna beseda, ampak le njen osnovni del brez končnice. Pri lematizaciji nam pomaga, da poznamo, kakšno oblikoskladenjsko označbo ima beseda, medtem ko pri stematizaciji tega ne potrebujemo. V angleščini se za stematizacijo najpogosteje uporablja Porterjev algoritem [37], uporabljamo pa lahko tudi druge algoritme.

4.3.3 Primerjava knjižnic: lematizacija

Odločili smo se, da postopek lematizacije v posameznih knjižnicah primerjamo po točnosti in hitrosti. Pri ocenjevanju točnosti algoritmov znotraj posameznih knjižnic smo se soočili s problemom, saj na žalost ne obstaja noben korpus z ročno določenimi lemami. Kot standard, s katerim smo nato primerjali druge knjižnice, smo zato vzeli lematizator Wordnet iz knjižnice NLTK, ki išče leme s pomočjo korpusa Wordnet, hkrati pa kot argument dobi tudi oblikoskladenjsko označbo besede (gre za tako imenovani 'srebrni standard' in ne zlati, saj je njegova pravilnost vprašljiva). Spodnja koda prikazuje izdelavo srebrnega standarda:

```
from nltk.corpus import treebank
from nltk.corpus import wordnet

#funkcija za izdelavo srebrnega standarda na podlagi lematizatorja s
pomočjo #knjižnice Wordnet
def wordnetLemmatizer(taggedWords):

    #inicializacija lematizatorja
    lemmatizer = WordNetLemmatizer()

    #ker oblikoskladenjske oznake v knjižnici Wordnet niso enake kot v
    #korpusu treebank, je potrebno narediti slovar preslikav Treebank oznak
    #v Wordnet oznake
    morphy_tag = {'NN':wordnet.NOUN, 'NNS':wordnet.NOUN,
                  'NNP':wordnet.NOUN, 'NNPS':wordnet.NOUN, 'JJ':wordnet.ADJ,
                  'JJR':wordnet.ADJ, 'JJS':wordnet.ADJ, 'VB':wordnet.VERB,
                  'VBD':wordnet.VERB, 'VBG':wordnet.VERB, 'VBN':wordnet.VERB,
                  'VBP':wordnet.VERB, 'VBZ':wordnet.VERB,'RB':wordnet.ADV,
                  'RBR':wordnet.ADV, 'RBS':wordnet.ADV}

    #lematizacija. Tu poskrbimo tudi za izpis v obliki seznama dvojic
    #(beseda, lema) ter pred samo lematizacijo preslikamo Treebank oznake
    #v wordnet oznake
    return [(word,lemmatizer.lemmatize(word, morphy_tag[pos]).lower()) if
           pos in morphy_tag else (word,lemmatizer.lemmatize(word).lower())
           for word, pos in taggedWords ]
```

```
#seznam tokeniziranih besed z oblikoskladenjskimi označbami iz korpusa
#Treebank
treebank_words = treebank.tagged_words()

#naredimo srebrni standard
lemmaList = wordnetLemmatizer(treebank_words)
```

Glede na empirične poskuse, ki so pokazali, da določene besede Wordnet lematizator iz knjižnice NLTK lematizira pravilno, medtem ko jih drugi lematizatorji ne, smo prišli do zaključka, da je uporaba tega lematizatorja kot srebrnega standarda upravičena, čeprav se seveda zavedamo dejstva, da tudi ta lematizator dela napake in kot standard ni tako zanesljiv kot ročno narejen korpus lem. Tako bralce opozarjamo, naj rezultate točnosti posameznih lematizatorjev jemljejo z rezervo, čeprav se nam zdi, da na ta način dobimo dokaj dobro oceno zanesljivosti, vsekakor pa vsaj dobro oceno podobnosti posameznih lematizatorjev.

Med seboj smo primerjali knjižnice NLTK, SpaCy, Pattern in TextBlob, saj knjižnica pyNLPl ne vsebuje lematizatorja. Prav tako smo dodali knjižnico Lemmagen [21], odprto-kodno knjižnico za lematizacijo, ki med 11 jezikovnimi modeli v različnih evropskih jezikih, ki temeljijo na RDR (*Ripple down rules*) pravilih, nudi tudi model za slovenski jezik, ki omogoča lematizacijo v slovenščini (čeprav smo v tem primeru zaradi lažje primerjave točnosti točnost lematizacije merili s pomočjo angleških besedil). Zanimivo pri tem lematizatorju je dejstvo, da za svoje delovanje ne potrebuje seznama besed, ampak le seznam pravil, ki je nastal s pomočjo strojnega učenja na učni množici (korpus z ročno določenimi lemami za vse besede). Pri RDR učenju se pravila inkrementalno dodajajo v seznam, čim se pojavijo novi primeri in odločitve. Ta nova pravila so lahko v nasprotju s starimi pravili, zato se poleg pravil v jezikovni model dodajajo tudi izjeme za stara pravila. Rezultat je hierarhično drevo pravil, ki določajo transformacijo besede v lemo.

V knjižnici Pattern sta dve funkciji za lematizacijo: funkcija 'parse', ki poleg lematizacije besedam dodeli tudi oblikoskladenjske označbe ter druge operacije, in funkcija 'stem', ki poleg stematizacije s pomočjo Porterjevega algoritma omogoča tudi iskanje lem. Knjižnica TextBlob vsebuje lematizator, ki tako kot knjižnica NLTK kot argument vzame oblikoskladenjsko oznako po standardu Wordnet. Zaradi te podrobnosti smo posumili, da knjižnica TextBlob uporablja NLTK-jev lematizator, čeprav to v dokumentaciji ni omenjeno, vendar pa bomo v nadaljevanju videli, da njen lematizator daje drugačne rezultate (tako po točnosti kot hitrosti) od knjižnice NLTK (srebrni standard), zato je njen lematizator očitno drugačen od NLTK-jevega, pa čeprav ravno tako uporablja Wordnet.

Za eksperiment in meritve smo uporabili korpus Treebank, ki vsebuje 100676 žetonov, točnost (*accuracy*) pa smo merili s pomočjo naslednje formule:

$$točnost = \frac{Število\ pravilno\ dodeljenih\ lem\ besedam}{Število\ vseh\ besed} \quad (4.1)$$

Točnost smo merili s pomočjo v NLTK integrirane funkcije 'accuracy' za meritev točnosti. Spodnja koda prikazuje implementacijo naših lematizatorjev iz knjižnic Lemmagen, Pattern SpaCy in TextBlob:

```
import lemmagen.lemmatizer
from lemmagen.lemmatizer import Lemmatizer
from pattern.text.en import parse
from spacy.en import English

#funkcija za Pattern lematizator v funkciji 'parse'. Potrebujemo branje
#korpusa po delčkih, drugače pride do izjeme 'object does not support item
#assignment' v knjižnici Pattern
def parseByChunks(l):
    list = []

    #lematiziramo po 100 besed velike delčke seznama
    for i in range(0,len(l),99):
        if len(l) - i < 99:
            parsed = parse(l[i:len(l)], chunks=False, tokenize=False,
                     tags=False, lemmata=True)
        else:
            parsed = parse(l[i:i + 99], chunks=False, tokenize=False,
                     tags=False, lemmata=True)

        #funkcija vrne seznam dvojic (beseda, lema)
        list.extend([(x.split("/")[0], (x.split("/")[2].lower())) for x in
                    parsed.split("\n")])
    return list


#lematizacija s pomočjo funkcije 'stem' v knjižnici Pattern. Funkcija vrne
#seznam dvojic (beseda, lema)
def patternLemmatizer(l):
    return [(word, stem(word, stemmer = LEMMA).lower()) for word in l]

#lematizator iz knjižnice Lemmagen. Funkcija vrne seznam dvojic (beseda,
#lema)
def lemmagenLemmanizer(l):
    lemmatizer = Lemmatizer(dictionary=lemmagen.DICTIONARY_ENGLISH)
    return [(word,lemmatizer.lemmatize(word).lower()) for word in l ]

#lematizator iz knjižnice SpaCy. Kot vidimo, gre za enak cevovod kot pri
#tokenizaciji, le da vrnemo seznam dvojic (beseda, lema)
def spacylemmatizer(l):

    #inicializacija cevovoda
    nlp = English()

    #v cevovodu s pomočjo opcijskega argumenta 'parse' izključimo
    #izdelavo stavčnih dreves, kar močno pospeši delovanje
    tokens = nlp(l.decode('unicode-escape'), parse=False, entity=False)
```

```
    listTokens = list(tokens)

    #naredimo seznam dvojic (beseda, lema)
    finallist = [(word.orth_, word.lemma_) for word in listTokens]
    return finallist

#lematizator knjižnice TextBlob
def textBlobLemmatizer(l):

    #tudi ta lematizator kot argument sprejme oblikoskladenjsko oznako na
    #podlagi standarda Wordnet, zato je tudi tu potreben slovar preslikav
    #Treebank oznak v Wordnet oznake
    morphy_tag = {'NN':wordnet.NOUN, 'NNS':wordnet.NOUN,
                  'NNP':wordnet.NOUN, 'NNPS':wordnet.NOUN,
                  'JJ':wordnet.ADJ, 'JJR':wordnet.ADJ, 'JJS':wordnet.ADJ,
                  'VB':wordnet.VERB, 'VBD':wordnet.VERB,
                  'VBG':wordnet.VERB, 'VBN':wordnet.VERB,
                  'VBP':wordnet.VERB, 'VBZ':wordnet.VERB,'RB':wordnet.ADV,
                  'RBR':wordnet.ADV, 'RBS':wordnet.ADV}

    #lematizacija in izpis seznama dvojic (beseda, lema)
    finallist = [(word, Word(word).lemmatize(morphy_tag[pos]).lower())  if
                  pos in morphy_tag else (word, Word(word).lemmatize()) for
                  word, pos in l]
    return finallist
```

Kar se tiče primerjave knjižnic, nam je največ težav povzročala knjižnica SpaCy, ki zaradi svojega cevovodnega sistema kot argument ne sprejema seznama besed, ampak le netokeniziran tekst. Problem smo rešili tako, da smo žetone v že tokeniziranem korpusu Treebank med seboj združili (Pythonova funkcija 'join') s presledki in to besedilo dali kot argument SpaCy-jevemu cevovodu. Cevovod je nato to besedilo na novo tokeniziral, vendar pa določeni žetoni niso bili isti kot v že tokeniziranem korpusu Treebank. Tako smo med seboj primerjali le enake žetone, ki jih je bilo 95695 od 100676, kar je verjetno malenkostno vplivalo na rezultat.

4.3.3.1 Rezultati meritev točnosti

	NLTK (srebrni standard)	**Pattern – parse**	**Pattern - stem**	**SpaCy**	**Lemmagen**	**TextBlob**
Točnost	1.0	0.968	0.871	0.934	0.939	0.948
Čas izvajanja (s)	3,4	16,57	3,91	4,8	3,02	3,95

Tabela 4.1: Rezultati meritev točnosti za posamezne lematizatorje

Rezultati meritev so predstavljeni v zgornji tabeli (Tabela 4.1). Kot vidimo, je največjo točnost dosegla Patternova funkcija parse, ki je kot informacijo vzela tudi oblikoskladenjsko označbo besede, s 97% točnostjo. Druga je bila knjižnica TextBlob, ki prav tako kot argument sprejme oblikoskladenjsko označbo, s 95% točnostjo. Zato domnevamo, da je ravno upoštevanje oblikoskladenjskih označb pripomoglo k večji točnosti od drugih knjižnic. SpaCy in Lemmagen sta dosegla podobno, tudi dokaj dobro, okoli 93% in 94% točnost, malce pa je razočarala Patternova funkcija stem, ki je dosegla le 87% točnost.

Morda malo neprimerna, a še vedno zanimiva je ocena točnosti stematizacijskih algoritmov, ki smo jih primerjali z istim srebrnim standardom kot lematizacijske algoritme zgoraj, čeprav se zavedamo, da je ocena nepoštena, saj je pričakovani rezultat korenjenja drugačen od rezultata lematizacije. Vendar nas je zanimalo, kolikšen delež besednih korenov se pri posameznih algoritmih za korenjenje dejansko ne razlikuje od lem. Točnost stematizatorjev smo merili s pomočjo spodnje funkcije:

```
from pattern.vector import stem, PORTER, LEMMA
from nltk.stem.lancaster import LancasterStemmer
from nltk.stem.porter import PorterStemmer
from nltk.stem.snowball import EnglishStemmer
from nltk.metrics import accuracy

#funkcija za izračun točnosti korenjenja
def stemmerAccuracy(l):

    #Lancasterjev algoritem
    Lancaster = [(word,LancasterStemmer().stem(word.lower())) for word in
    l]

    #Porterjev algoritem v knjižnici NLTK
    Porter = [(word,PorterStemmer().stem(word.lower())) for word in l]

    #algoritem snežne kepe
    Snowball = [(word,EnglishStemmer().stem(word.lower())) for word in l]

    #Porterjev algoritem v knjižnici Pattern
    PatternPorter = [(word, stem(word, stemmer = PORTER).lower()) for word
    in l]

    #izpis točnosti s pomočjo v NLTK integrirane funkcije za meritev
    #točnosti. Algoritme primerjamo s srebrnim standardom
    print accuracy(lemmaList, Lancaster)
    print accuracy(lemmaList, Porter)
    print accuracy(lemmaList, Snowball)
    print accuracy(lemmaList, PatternPorter)
```

Tabela 4.2 prikazuje rezultate meritev.

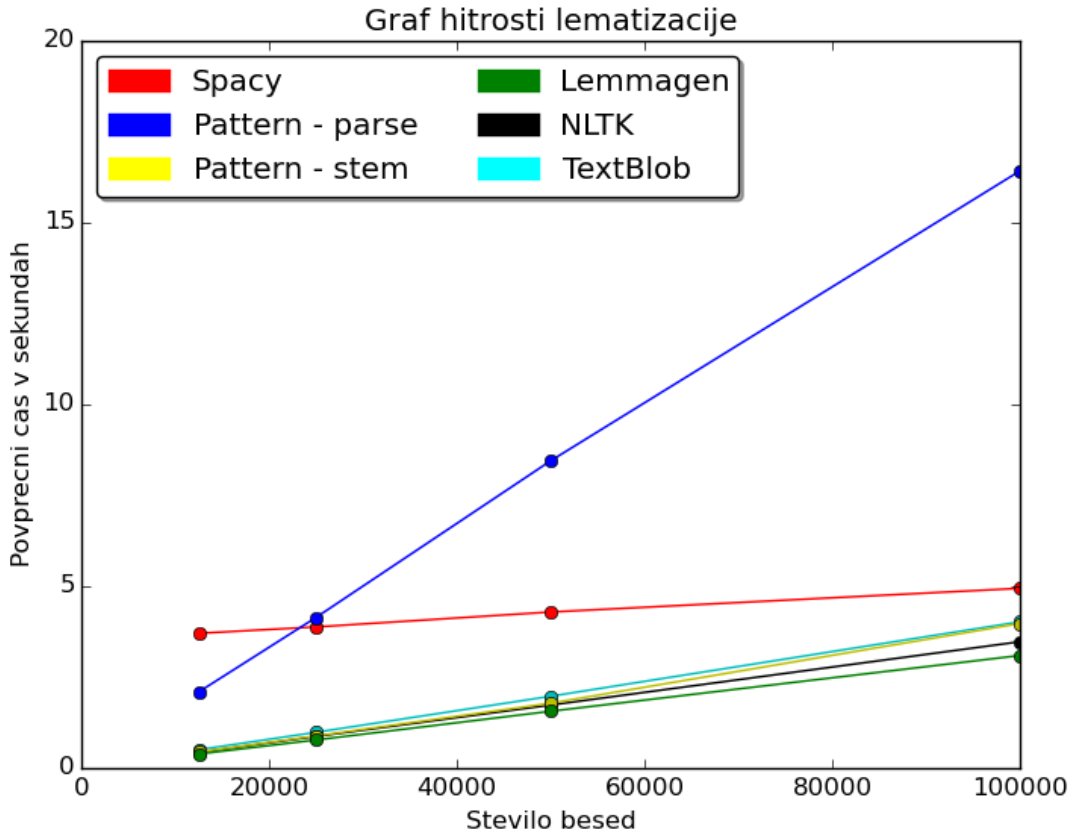
	NLTK (srebrni standard)	**Pattern - stem (Porterjev algoritem)**	**NLTK – Porterjev algoritem**	**NLTK – Lancasterjev algoritem**	**NLTK – Algoritem snežne kepe**
Točnost	1.0	0.783	0.777	0.687	0.783
Čas izvajanja (s)	3,4	4,1	5,84	25,74	4,93

Tabela 4.2: Rezultati meritev točnosti za posamezne stematizacijske algoritme

Vidimo, da Porterjev algoritem v obeh knjižnicah ter algoritem snežne kepe v NLTK dosegajo okoli 78% točnost, če jih primerjamo z lematizacijskim srebrnim standardom, kar pomeni, da je 78% korenov besed enakih lemi te besede. Lancasterjev algoritem [41] je dosegel precej slabšo točnost (69%), vendar pa to nujno ne pomeni, da je ta algoritem za stematizacijo slab, pomeni le, da so koreni, ki jih dodeli besedam, drugačni od lem teh besed.

4.3.3.2 Primerjava hitrosti knjižnic

Knjižnice in njihove algoritme smo med seboj primerjali tudi po hitrosti. Hitrost smo merili na istem korpusu kot točnost, tj. korpusu Treebank. Hitrost je bila za vse knjižnice izmerjena 10-krat na seznamih besed, ki so obsegali 12500, 25000, 50000 in 100000 prvih besed iz korpusa. Nekaj problemov nam je povzročala knjižnica SpaCy, kjer je operacija lematizacije neločljivo povezana z drugimi operacijami nad besedilom in kot vhod sprejme le besedilo in ne seznama besed. Da bi dobili primerljivo velikost besedila, smo seznam žetonov združili s presledki (Pythonova funkcija 'join') in knjižnici SpaCy kot argument dali to besedilo. Koda za primerjavo hitrosti knjižnic je priložena v dodatku diplomske naloge. Slika 4.2 prikazuje rezultate meritev.



Slika 4.2: Primerjava hitrosti lematizacije

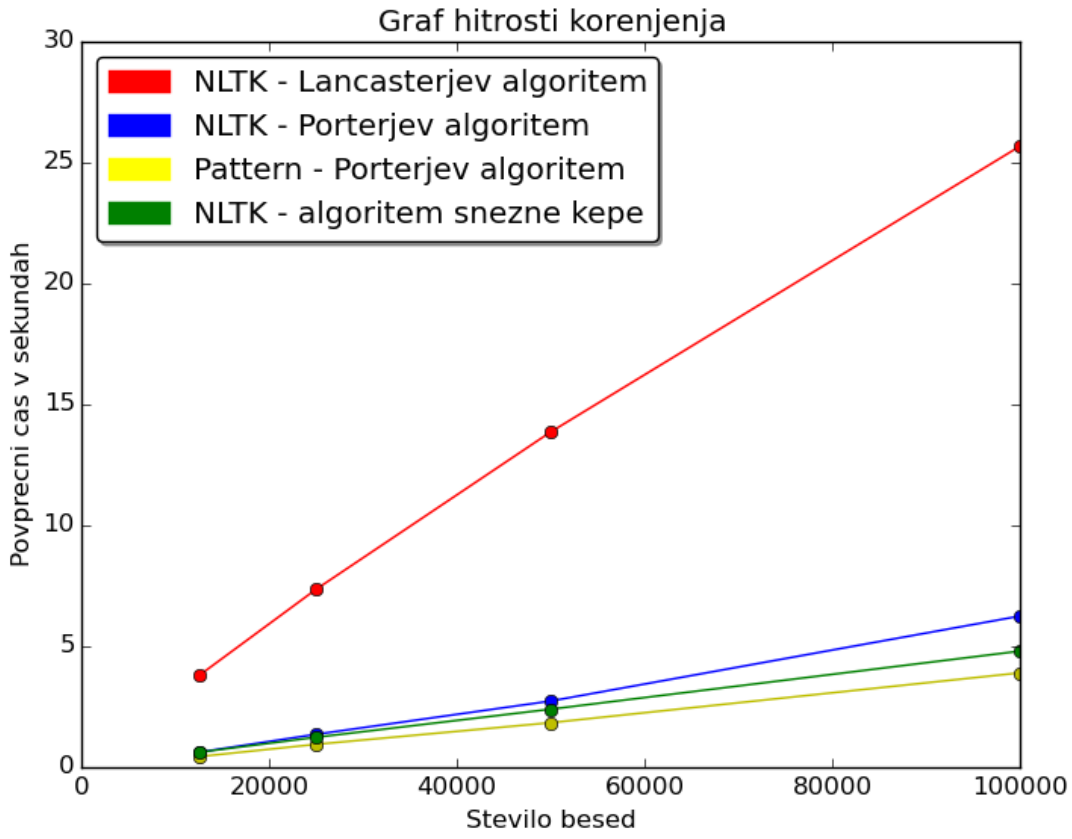
Za začetek lahko opazimo, da so vse funkcije linearne, kar pomeni, da čas narašča linearno glede na število besed. Vendar pa je očitno, da časovna zahtevnost Patternove funkcije 'parse' z večanjem števila besed narašča precej hitreje kot pri drugih funkcijah, tako da je za korpus s 100000 besedami porabila več kot trikrat več časa kot druge funkcije.

Najpoložnejšo linijo ima knjižnica SpaCy, ki za lematizacijo korpusa 100000 besed porabi le kakšno sekundo več kot pa za lematizacijo korpusa 12500 besed. Očitno je, da ta knjižnica potrebuje nekaj časa za inicializacijo njenega cevovoda, vendar pa se z večanjem korpusa ta začetni zaostanek sčasoma zmanjšuje. Tudi drugi štirje lematizatorji, tj. knjižnica Lemmagen, NLTK-jev Wordnet lematizator, Patternova funkcija stem in lematizator v knjižnici TextBlob, so vsi opravili s seznamom 100000 besed v manj kot 5 sekundah. Pri korpusih z manj kot 25000 besedami razlik med njimi ni, pri večjih korpusih pa opazimo majhne razlike v hitrosti. Standardni odkloni teh knjižnic pri meritvah so se merili v stotinkah (NLTK 6 stotink, funkcija 'stem' v knjižnici Pattern 3 stotinke, Lemmagen 0.7 stotinke in TextBlob 6 stotink). Očitno je, da je NLTK-jev lematizator s pomočjo korpusa Wordnet, ki smo ga pri merjenju točnosti vzeli za srebrni standard, dobra izbira za izvedbo lematizacije tudi zaradi hitrosti, saj

se je izkazal za drugega najhitrejšega. To velja pod pogojem, da imamo na voljo tudi oblikoskladenjske označbe besed. Če teh nimamo, je zelo dobra izbira tudi odprtokodna knjižnica Lemmagen, ki je še za malenkost hitrejša in prav tako dosega dobro točnost, in to celo brez oblikoskladenjskih označb za besede. Skoraj enako hitra kot NLTK-jev lematizator ter knjižnici Lemmagen in TextBlob je tudi Patternova funkcija 'stem', ki pa je na žalost precej manj točna, zato pri lematizaciji morda ni prava izbira.

4.3.4 Primerjava hitrosti korenjenja

Ker nas je zanimalo, koliko je operacija stematizacije kompleksnejša od operacije korenjenja, smo na enako velikih seznamih besed izmerili tudi hitrost algoritmov za korenjenje.



Slika 4.3: Prikaz hitrosti korenjenja

Rezultati, ki jih prikazuje zgornji graf (Slika 4.3), so presenetljivi, saj smo pričakovali, da bodo algoritmi za korenjenje hitrejši od algoritmov za lematizacijo. A dejansko niso po teh meritvah algoritmi za korenjenje nič hitrejši od algoritmov za lematizacijo, ampak sta NLTK-jeva Porterjev algoritem ter algoritem snežne kepe pri seznamu s 100000 besedami celo za

sekundo počasnejša od lematizacije s pomočjo Wordneta (tudi pri meritvah za ta dva algoritma so se standardni odkloni merili v stotinkah sekunde). Lancasterjev algoritem je po drugi strani še veliko počasnejši, čas pa z dolžino seznama strmo narašča. Pri 100000 besedah potrebuje za korenjenje celo več kot 25 sekund (standardni odklon pri meritvah za ta stematizator je 0.7 sekunde). Najhitrejša je Patternova implementacija Porterjevega algoritma, ki obdela seznam 100000 besed v času okoli 4 sekunde (standardni odklon je ena desetinka sekunde).

4.4 Oblikoskladenjsko označevanje

Pri oblikoskladenjskem označevanju posameznim besedam v stavku pripišemo slovnične oznake, stavke pa transformiramo v sezname dvojic beseda/oznaka, kjer je oznaka samostalnik, glagol, pridevnik ali kakšna druga slovnična oznaka. Obstaja več konvencij za označevanje besed, v angleščini pa je najpogostejše tako imenovano poenostavljeno označevanje, ki temelji na množici oznak Penn Treebank ('Penn Treebank II tag-set'). Pripisovanje slovničnih oznak besedam daje določeno semantično informacijo o besedi. Tako velja, da se samostalniki načeloma nanašajo na ljudi, stvari, kraje in koncepte, glagoli pa se nanašajo na dejanja in dogodke in ponavadi izražajo relacijo med več samostalniki. Pridevniki opisujejo samostalnike (velikokrat izražajo sentiment do določenega samostalnika), prislovi pa običajno opisujejo okoliščine, v katerih poteka dejanje.

Lingvisti pripisujejo slovnične kategorije na podlagi morfoloških (oblikovnih), sintaktičnih in semantičnih indicev. Že sama oblika besede nam včasih pove, za kakšno besedo gre. Na primer, končnice 'ness' ali 'ment' so v angleščini zelo močan indikator, da gre za samostalnik. Včasih besedo lažje kategoriziramo na podlagi konteksta (tu gre ponavadi za sintaktične namige). Na primer, pridevnik je običajno vedno pred samostalnikom in za glagolom, tako da če vemo, da je določena beseda samostalnik in neka druga beseda glagol, je precej verjetno, da je beseda med njima pridevnik. Semantični kriteriji za kategorizacijo so v moderni lingvistiki obravnavani s kopico dvomov, saj jih je težko formalizirati, vendar pa še vedno velja, da nam ravno semantične definicije povedo največ. Tako je najboljša definicija samostalnika še vedno semantična: »Ime osebe, stvari ali kraja.« [24]

4.4.1 Opis označevalnikov

Knjižnica NLTK vsebuje veliko označevalnikov, večina od njih pa je učljivih, kar pomeni, da jih s pomočjo množice že vnaprej označenih stavkov (učna množica) lahko naučimo, da označujejo nove stavke. Označevalnike v tej knjižnici lahko tudi povežemo v verigo, tako da v primeru, ko prvi označevalnik ne more označiti določene besede, pokličemo drugi

označevalnik v verigi. Najosnovnejši označevalnik v knjižnici NLTK je privzeti označevalnik, kjer vse besede označimo z isto oznako. Ta je uporaben kot pomožni označevalnik, ki je pripet na koncu verige označevalnikov, hkrati pa nam lahko služi kot standard ('baseline'), na podlagi katerega lahko ocenjujemo druge označevalnike [3].

Drugi najpreprostejši označevalnik v knjižnici NLTK je tako imenovani unigramski označevalnik. Z unigramom ponavadi mislimo na posamezen žeton. Unigramski označevalnik tako pri označevanju besed kot kontekst upošteva le besedo, ki jo želi označiti. Torej gre za označevalnik, ki označuje s pomočjo konteksta, kontekst pa je le ena sama beseda.

Pri učenju označevalnik naredi kontekstualni model s pomočjo seznama označenih besed, ki mu pomaga pri pripisovanju oznak drugim besedam. Za kontekst unigramski označevalnik vzame le eno besedo. Sam model na podlagi seznama označenih stavkov izračuna frekvenco posameznih označb za določen kontekst (v tem primeru je kontekst le ena beseda, ki je lahko v istem besedilu hkrati pridevnik, samostalnik ali kaj drugega). Označba z najvišjo frekvenco je shranjena v modelu in pripisana novim istim besedam.

Poleg unigramskih označevalnikov sta v NLTK vgrajena tudi razreda z bigramskim (razred 'BigramTagger') in trigramskim (razred 'TrigramTagger') označevalnikom. Bigramski označevalnik kot kontekst vzame besedo, ki jo želi označiti, in besedo pred njo, trigramski pa besedo, ki jo želi označiti, in prejšnji dve besedi. Ta dva označevalnika izboljšata točnost dodeljevanja oznak zato, ker je oznaka besede velikokrat odvisna od konteksta, v katerem se nahaja. Na primer »tag« je v angleščini lahko samostalnik 'oznaka' ali pa glagol 'označiti'. Ideja n-gramskih označevalnikov je, da nam prejšnje besede lahko pomagajo pri odločitvi, kakšno oznako naj dodelimo tem dvoumnim besedam. Vendar se sama po sebi bigramski in trigramski označevalnik obneseta zelo slabo (bigramski označevalnik pri empiričnih poskusih dosega okoli 11% točnost, trigramski pa 7%), med drugim tudi zaradi tega, ker ne moreta dodeliti oznak vsem prvim besedam v stavku. Dobro pa se obneseta v kombinaciji z drugimi označevalniki, če ju vključimo v verigo z drugimi označevalniki.

Slika 4.4: Veriga označevalnikov: Zelena puščica predstavlja uspešen pripis oblikoskladenjske oznake, rdeča pa neuspešen pripis oznake

Na zgornji sliki (Slika 4.4) vidimo, kako deluje veriga označevalnikov. Najprej poskuša oznako dodeliti trigramski označevalnik, če mu ne uspe, besedo (žeton) prepusti bigramskemu, ki v primeru, da mu prav tako ne uspe dodeliti oznake, besedo prepusti unigramskemu. Ta besedo v primeru, da mu ne uspe dodeliti oznake, prepusti privzetemu označevalniku, ki dodeli najpogostejšo oznako v besedilu.

NLTK poleg zgoraj navedenih vsebuje še nekaj zanimivih označevalnikov:

- Označevalnik s pomočjo regularnih izrazov, ki kot argument dobi seznam dvojic, od katerih je prva od dvojice regularni izraz, druga pa oznaka. Na primer, besede, ki se končajo v angleščini z 'ing' (regularni izraz r'.*ing$'), so ponavadi glagoli.

- Označevalnik na podlagi začetnic in končnic besed, ki kot kontekst vzame začetek ali konec besede in se nauči označevanja le na podlagi slednjega, ne pa cele besede. Privzeti argumenti za ta označevalnik so, da mora biti beseda dolga vsaj 5 znakov, gleda pa zadnje tri črke v besedi. Drugače dodeli oznako 'none'. Te parametre lahko spreminjamo.

- Brillov označevalnik [38], ki temelji na transformacijah. Logika delovanja je dokaj preprosta: na začetku uganemo oznako, nato se vrnemo nazaj in popravimo vse napačne oznake. Te napačne oznake se popravljajo na podlagi seznama pravil, ki ga ta

označevalnik sproti nadgrajuje. Ta pravila imajo ponavadi strukturo 'zamenjaj oznako 1 z oznako 2 v kontekstu K' (na primer 'zamenjaj samostalnik z glagolom, če je prejšnja beseda veznik') in so ocenjena po naslednji formuli :

$$Ocena = \check{S}tevilo\ pravilnih\ popravkov - \check{s}tevilo\ nepravilnih\ popravkov \quad (4.2)$$

Ta pravila so zanimiva zaradi tega, ker so lingvistično interpretabilna in nam dajo določeno informacijo o obliki in pomenu besedila. Nato na podlagi teh pravil označevalnik transformira prej predpisane oznake.

- TnT (Trigrams 'n' Tags) označevalnik [40] je statističen označevalnik, ki temelji na Markovskem modelu drugega reda. Ta označevalnik hrani frekvenčne porazdelitve besed glede na učne podatke. Te frekvenčne distribucije štejejo tako unigrame kot tudi bigrame in trigrame. Med označevanjem se te frekvence upoštevajo pri izračunu verjetnosti, da ima posamezna beseda ali sklop besed določeno označbo. Ta označevalnik dosega dobre rezultate glede točnosti (89%), učenje je hitro, vendar pa je označevanje dokaj počasno.

- Za označevanje v angleščini lahko uporabimo tudi knjižnico Wordnet, ki je ogromen slovar angleškega jezika in vsebuje veliko besed, ki jih običajna učna množica ne vsebuje, zato ga lahko uporabimo za označevanje neznanih besed. V množici sopomenk tako pogledamo, katera je najbolj verjetna oznaka za iskano besedo in jo dodelimo besedi. Ker so oznake v Wordnetu drugačne kot v drugih korpusih, jih je treba preslikati v nove. Poleg tega Wordnet vsebuje le štiri splošne oznake, medtem ko jih korpus 'Treebank' vsebuje 36, zato ta označevalnik ni zadovoljiv kot samostojen označevalnik, se pa dobro obnese kot zadnji v verigi med seboj povezanih označevalnikov.

- Označevalnik, ki označuje besede s pomočjo klasifikacije. Ta označevalnik deluje tako, da najprej iz besed izlušči pomembne značilke in nato na podlagi teh značilk določi razred (oblikoskladenjsko oznako). Ta označevalnik je preprost za uporabo in dosega zelo dobre rezultate. Privzeto klasificira s pomočjo naivnega Bayesovega klasifikatorja in je pogosto najbolj točen označevalnik, vendar pa je tudi eden od najpočasnejših. Pri klasifikatorju s pomočjo klasifikacije lahko sami izberemo algoritem za klasificiranje in značilke, ki jih želimo uporabiti pri klasifikaciji. Prav tako mu lahko pripnemo druge klasifikatorje, ki klasificirajo v primeru, da je verjetnost najbolj verjetne oznake manjša od neke definirane mejne verjetnosti.

Knjižnica Pattern ima vgrajen Brillov označevalnik, ki deluje po enaki logiki kot Brillov označevalnik v knjižnici NLTK. Že v prejšnjem poglavju smo omenili, da tudi knjižnica TextBlob za pripis oblikoskladenjskih označb privzeto uporablja kar označevalnik knjižnice Pattern, možno pa je uporabiti tudi privzeti označevalnik iz knjižnice NLTK, torej označevalnik s pomočjo maksimalne entropije.

Tudi knjižnica SpaCy v svojem cevovodu vsebuje oblikoslovni označevalnik, ki načeloma sodi v skupino klasifikacijskih označevalnikov, ki temeljijo na nevronski mreži ('averaged perceptron') [1]. Takšni označevalniki se učijo z pomočjo iterativnega prilagajanja uteži na povezavah med nevroni učni množici. Tudi v tej knjižnici je označevalnik neločljivo povezan z drugimi operacijami.

4.4.2 Primerjava označevalnikov po točnosti

SpaCyjev in Patternov označevalnik pripisujeta označbe iz množice označb Penn Treebank II tag-set, NLTK-jevi označevalniki pa so narejeni tako, da jih lahko naučimo, da dodeljujejo označbe iz katerekoli množice. Da bi bili označevalniki med seboj primerljivi, smo vse NLTK-jeve označevalnike učili na korpusu Treebank z že oblikoskladenjsko označenimi besedami, in sicer smo prvih 9/10 korpusa uporabili za učno množico, zadnjo desetino pa za testiranje. Po drugi strani sta bila SpaCyjev in Patternov označevalnik naučena že vnaprej (Patternov na neznanem korpusu 100000 angleških besed, SpaCyjev pa na plačljivem korpusu Treebank-2, ki vsebuje 1 milijon že označenih besed). Kot zlati standard, s katerim smo primerjali oznake, ki so jih dodelili označevalniki, smo uporabili ročno označeni korpus Penn Treebank (pri NLTK-jevih označevalnikih le zadnjo desetino, pri ostalih pa celega). Točnost smo merili po naslednji formuli:

$$\text{točnost} = \frac{\check{S}tevilo\ pravilno\ dodeljenih\ oblikoskladenjskih\ označb\ besedam}{\check{S}tevilo\ vseh\ besed} \tag{4.3}$$

Pri NLTK označevalnikih smo tako za izračun točnosti uporabili kar v označevalnike vgrajeno funkcijo 'evaluate', ki računa točnost po zgoraj omenjeni formuli. Programska koda za označevanje s pomočjo označevalnikov iz knjižnice SpaCy in Pattern je zelo podobna kodi za lematizacijo s pomočjo teh dveh knjižnic, saj pri obeh operacijo izvede ista funkcija (v knjižnici Pattern funkcija 'parse', v knjižnici SpaCy pa cevovod 'nlp') z malce drugačnimi argumenti, zato kode ne bomo navedli, temveč jo bomo priložili v dodatku diplomske naloge. Poglejmo si programsko kodo za učenje in evaluacijo NLTK označevalnikov:

```
import nltk
import nltk.tag
from nltk.tag import RegexpTagger
from nltk.tag import AffixTagger
from nltk.tag.brill import fntbl37
from nltk.tag.brill_trainer import BrillTaggerTrainer
from nltk.tag import tnt
from nltk.tag.sequential import ClassifierBasedPOSTagger
from nltk.classify import MaxentClassifier

#računanje točnosti za različne NLTK označevalnike
#učenje in evaluacija unigramskega označevalnika
def unigramtaggerAccuracy():
    unigram_tagger = nltk.UnigramTagger(train_sents)
    return unigram_tagger.evaluate(test_sents)

#učenje in evaluacija bigramskega označevalnika
def bigramtaggerAccuracy():
    bigram_tagger = nltk.BigramTagger(train_sents)
    return bigram_tagger.evaluate(test_sents)

#učenje in evaluacija verižnega označevalnika, ki ima v verigi pripete
#3 označevalnike: privzeti, unigramski in bigramski označevalnik
def chainedtaggerAccuracy():
    t0 = nltk.DefaultTagger('NN')
    t1 = nltk.UnigramTagger(train_sents, backoff=t0)
    t2 = nltk.BigramTagger(train_sents, backoff=t1)
    return t2.evaluate(test_sents)

#inicializacija in evaluacija označevalnika s pomočjo regularnih izrazov
def regextaggerAccuracy():

     #definiranje vzorcev za označevanje
     patterns = [
        (r'^\d+$', 'CD'),
        (r'.*ing$', 'VBG'),
        (r'.*ment$', 'NN'),
        (r'.*ful$', 'JJ')
        ]
    tagger = RegexpTagger(patterns)
    return tagger.evaluate(test_sents)

#inicializacija in evaluacija označevalnika s pomočjo končnic
def affixtaggerAccuracy():

    #argument 'affix_length'=-2' določa, da gledamo zadnji dve črki besede
    tagger = AffixTagger(train_sents, affix_length=-2)
    print tagger.evaluate(test_sents)

#Brillov označevalnik - za osnovo vzamemo verižni označevalnik, ki vsebuje
#unigramski, bigramski in privzeti označevalnik
def brilltaggerAccuracy():
    t0 = nltk.DefaultTagger('NN')
    t1 = nltk.UnigramTagger(train_sents, backoff=t0)
    t2 = nltk.BigramTagger(train_sents, backoff=t1)

    #definiranje vzorca za učenje pravil
    templates = fntbl37()
    trainer = BrillTaggerTrainer(t2, templates)
```

```
    brill_tagger = trainer.train(train_sents)
    return brill_tagger.evaluate(test_sents)

#učenje in evaluacija tnt označevalnika
def tnttaggerAccuracy():
    tnt_tagger = tnt.TnT()
    tnt_tagger.train(train_sents)
    print tnt_tagger.evaluate(test_sents)

#učenje in evaluacija označevalnika s pomočjo klasifikacije - privzeto
#klasificira s pomočjo naivnega Bayesa
def classifiertaggerAccuracy():
    tagger = ClassifierBasedPOSTagger(train=train_sents)
    print tagger.evaluate(test_sents)

#učenje in evaluacija označevalnika s pomočjo klasifikacije -
#klasificiranje s pomočjo maksimalne entropije
def maxenttaggerAccuracy():
    tagger = ClassifierBasedPOSTagger(train=train_sents,
             classifier_builder=MaxentClassifier.train)
    print tagger.evaluate(test_sents)
```

Rezultate meritev nam prikaže Tabela 4.3.

	SpaCy - Perceptron	**Pattern – Brill**	**NLTK - unigram**	**NLTK – verižni**	**NLTK - Brill**	**NLTK - tnt**	**NLTK – naive Bayes**	**NLTK - maxent**
Točnost	0.936	0.851	0.864	0.891	0.895	0.881	0.932	0.927
Čas izvajanja(s)	6,32	14,67	5,79	6,71	8,68	/	249,85	131,44

Tabela 4.3: Prikaz točnosti oblikoskladenjskih označevalnikov

Kot vidimo, je SpaCyjev označevalnik dosegel največjo točnost (skoraj 94%), blizu pa sta mu prišla le še NLTK-jeva označevalnika, ki temeljita na klasifikaciji (označevalnika z algoritmom naivnega Bayesa ter algoritmom za iskanje maksimalne entropije sta dosegla približno 93%). Dobro se je obnesel NLTK-jev verižni označevalnik (89% točnost), kjer smo v verigo spravili unigramski (ki je sam dosegel 86,4% točnost), bigramski ter privzeti označevalnik, ki je vsem neznanim besedam, ki jim druga dva označevalnika v verigi nista znala prepisati oznake, pripisal oznako za samostalnik (NN). Točnosti označevalnikov knjižnice TextBlob nismo merili, saj ta knjižnica privzeto uporablja že naučeni označevalnik iz knjižnice Pattern, lahko pa uporabimo tudi NLTK-jev privzeti označevalnik s pomočjo

maksimalne entropije, kar pomeni, da meritve točnosti teh dveh označevalnikov veljajo tudi za knjižnico TextBlob.

Brillov označevalnik smo implementirali tako, da smo za začetni označevalnik vzeli verižni označevalnik (Slika 4.4), Brillov algoritem pa se je nato naučil skupek pravil, ki so popravila oznake, ki jih je verižni označevalnik napačno dodelil. Na ta način smo točnost verižnega označevalnika izboljšali še za slabo polovico odstotka in dobili 89,5% točnost. Malce nas je razočaral Patternov Brillov označevalnik, ki je dosegel le 85% točnost. Mogoče je vzrok za to v učni množici, ki se morda preveč razlikuje od testne, da bi algoritem lahko dosegel boljši rezultat, morda pa je problem v sami implementaciji algoritma. Solidno se je obnesel tudi označevalnik TNT, ki je dosegel 88% točnost, vendar pa nas je pri njem motila njegova nestabilnost, počasnost in slaba implementacija, saj je pri označevanju večjih korpusov večkrat prišlo do napake prekoračitve rekurzivnega limita.

Kar se tiče časov izvajanja, se vsi nanašajo na označevanje celotnega korpusa Treebank (ki vsebuje nekaj več kot 100000 besed), čeprav smo pri nekaterih označevalnikih točnost merili le na zadnji desetini korpusa (prvih 9/10 korpusa smo uporabili za označevanje). Časa izvajanja označevalnika TnT nam ni uspelo izmeriti, saj ta pri označevanju prevelikih korpusov vrže napako zaradi presežene maksimalne globine rekurzije. Po drugi strani nam je uspelo izmeriti njegovo točnost na zadnji desetini korpusa.

Poleg označevalnikov v tabeli smo testirali tudi bigramski označevalnik, ki ni primeren za samostojno rabo, saj je avtonomno dosegel le 13% točnost, se je pa dobro obnesel znotraj verige označevalnikov. Označevalnik s pomočjo regularnih izrazov je zahteven za implementacijo, saj je treba ročno definirati veliko množico pravil, kar zahteva dobro poznavanje angleškega oblikoslovja. S pomanjkljivo množico (naša je obsegala le 4 pravila), smo dosegli le okoli 0,5% točnost, zato ga nismo uporabili niti v verigi označevalnikov. Označevalnik s pomočjo končnic besed je z malce prilagoditvami privzetih parametrov (najboljše rezultate je dalo gledanje le zadnjih dveh črk v besedi) dosegel 32% točnost.

4.4.3 Primerjava oblikoskladenjskih označevalnikov po hitrosti

Pri naslednjem testu smo preverili hitrost označevalnikov. Ni nas zanimala hitrost učenja, ampak le hitrost označevanja besed že naučenih označevalnikov. In če v prejšnjem poglavju nismo merili točnosti označevalnika iz knjižnice TextBlob, saj ta knjižnica privzeto uporablja implementacijo že naučenega Patternovega označevalnika, pa smo se odločili, da njegovo hitrost vseeno izmerimo, saj je v tej knjižnici označevalnik implementiran kot metoda razreda 'TextBlob', v knjižnici Pattern pa kot del funkcije 'parse' (knjižnica Pattern sicer vsebuje tudi funkcijo 'tag', ki omogoča oblikoskladenjsko označevanje, vendar nam pogled v izvorno kodo

razkrije, da ta funkcija kliče funkcijo 'parse'). Razlika v implementaciji istega označevalnika pa bi lahko vplivala na hitrost. Poglejmo si še kodo, s pomočjo katere smo implementirali označevalnik knjižnice 'TextBlob':

```
from textblob import TextBlob
#označevalnik knjižnice TextBlob

def textBlobtagger(s):
    return TextBlob(s).tags
```

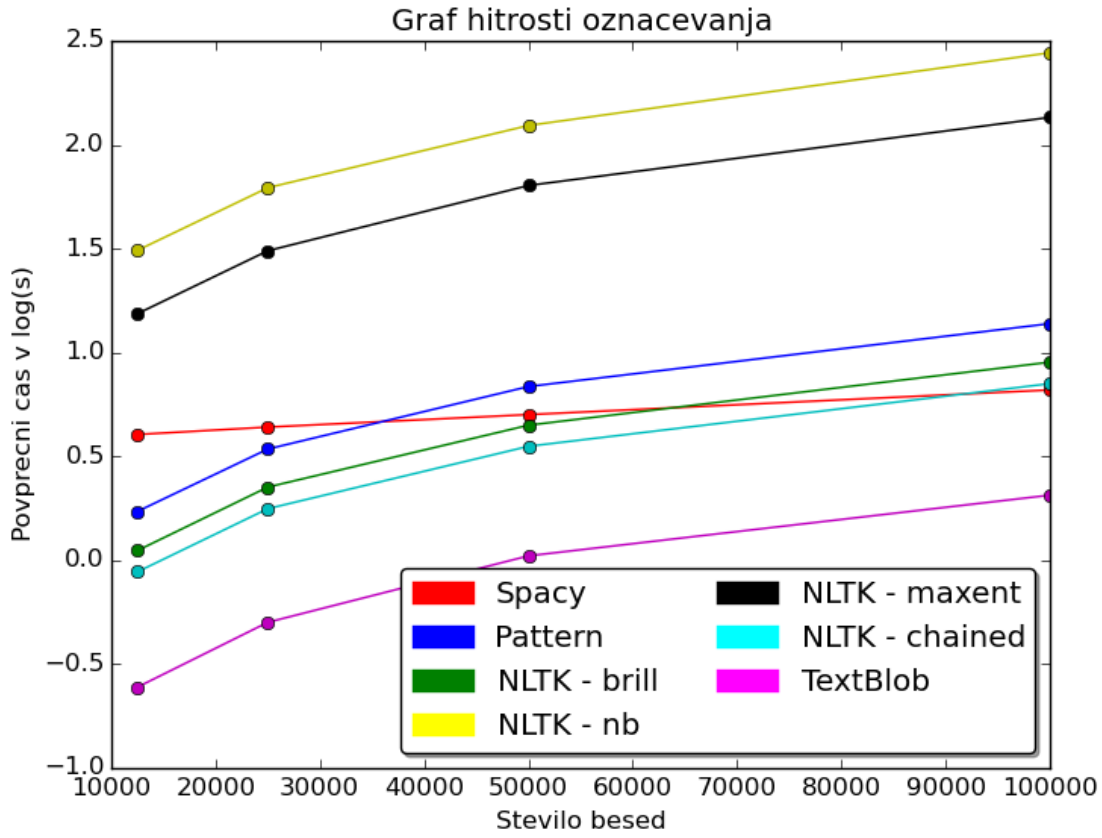
Kot vidimo, je implementacija zelo preprosta, saj pokličemo le metodo razreda 'TextBlob', ki je ovojni razred za besedilni niz (*string*). Konstruktor tega razreda kot argument sprejme le besedilni niz, ne pa seznama besed, zato naša metoda poleg označevanja izvede tudi tokenizacijo (poskusi, kjer smo naredili nov 'TextBlob' objekt za vsako besedo iz seznama, so pokazali, da je to dejansko počasneje, kot če poleg označevanja izvedemo še tokenizacijo celotnega korpusa).

Tako kot pri lematizaciji smo hitrost preverjali na štirih različno velikih besedilih: prvih 100000 besed korpusa Treebank, prvih 50000 besed korpusa Treebank, prvih 25000 besed korpusa Treebank in prvih 12500 besed korpusa Treebank. Knjižnici SpaCy in TextBlob, kjer je označevanje neločljivo od tokenizacije, smo kot argument dali besede, združene s presledki (Pythonova funkcija 'join').

Rezultati so predstavljeni na grafu (Slika 4.5). Zaradi boljše preglednosti smo tu namesto linearne skale za čas uporabili logaritemsko skalo. Za začetek lahko opazimo, da se označevalniki po hitrosti na grobo delijo v tri skupine. Oba na kategorizaciji temelječa označevalnika iz knjižnice NLTK sta veliko počasnejša od vseh ostalih označevalnikov (ne pozabimo, da je za čas uporabljena logaritemska skala). Klasifikacijski označevalnik, ki temelji na maksimalni entropiji, se je presenetljivo izkazal za hitrejšega od naivnega Bayesa), kar je verjetno posledica tega, da pri teh meritvah nismo ustvarili lastnega označevalnika z maksimalno entropijo kot pri meritvah točnosti, ampak smo uporabili kar privzeti označevalnik v NLTK, ki je tudi označevalnik s pomočjo maksimalne entropije (pri merjenju točnosti ta ni prišel v poštev, saj se privzeto uči na celotnem korpusu Treebank, zato ga ni bilo mogoče testirati na istem korpusu). Ta označevalnik je verjetno vsaj delno hitrostno optimiziran, a zato manj prilagodljiv.

Označevalnik, ki izstopa v drugi hitrejši skupini označevalnikov je SpaCyjev označevalnik. Kot vidimo, nanj velikost korpusa ne vpliva tako močno kot na druge označevalnike. Ta označevalnik potrebuje kar nekaj časa za inicializacijo, vendar pa se že pri korpusu z okoli 90000 tisoč besed izkaže za boljšo izbiro od drugih označevalnikov v tej skupini, če nas

zanima le hitrost. Pri manjših korpusih je vredno uporabiti Brillov ali verižni označevalnik, ki sta hitrejša, razlika med njima v hitrosti pa je skoraj zanemarljiva.



Slika 4.5: Primerjava hitrosti označevanja

Največje presenečenje pa je seveda implementacija Patternovega označevalnika v knjižnici TextBlog, ki je za celoten hitrostni razred hitrejši od vseh ostalih označevalnikov, kljub temu da poleg samega označevanja izvede tudi tokenizacijo. Očitno je Patternova implementacija istega označevalnika v funkciji 'parse' veliko počasnejša od implementacije tega označevalnika znotraj razreda 'TextBlob', kljub temu da smo s pomočjo opcijskih argumentov v funkciji 'parse' izključili vse dodatne operacije, kot sta tokenizacija in lematizacija.

Če sedaj pogledamo, kateri označevalnik je najbolje uporabiti, hitro vidimo, da oba na kategorizaciji temelječa označevalnika kljub svoji točnosti odpadeta zaradi svoje počasnosti, saj nam SpaCyjev perceptron nudi za skoraj odstotek boljšo točnost pri veliko hitrejšem izvajanju. Če nam je hitrost pomembnejša od točnosti, je najboljša izbira knjižnica TextBlob, vendar pa ima ta označevalnik slabo točnost (okoli 85%). V primeru, da želimo oblikoskladenjske oznake pripisati slovenskemu besedilu, nam ne preostane drugega, kot da

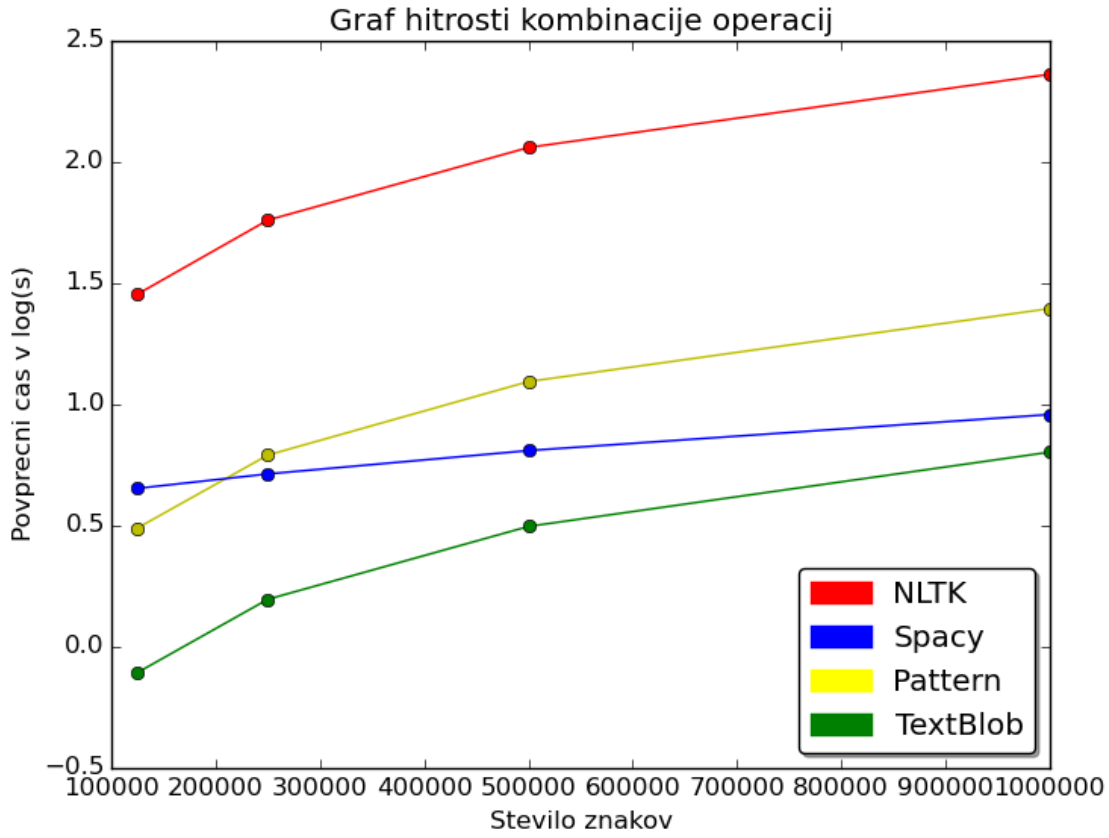
uporabimo enega od klasifikatorjev knjižnice NLTK, saj so edino ti učljivi. V tem primeru priporočamo uporabo označevalnika s pomočjo klasifikacije (klasifikator s pomočjo naivnega Bayesa se je izkazal za točnejšega – njegova točnost je 93%), če nam ni pomembna hitrost, ali pa Brillov označevalnik, ki je hiter in dosega tudi dokaj dobro točnost (89,5%).

4.5 Primerjava hitrosti kombinacije operacij

Že večkrat smo omenili, da knjižnica SpaCy vsebuje cevovod, ki smiselno združuje razne operacije za obdelavo naravnega jezika. Kombiniranje temeljnih operacij, kot so tokenizacija, lematizacija in oblikoskladenjsko označevanje, je smiselna, saj v večini primerov obdelave naravnega jezika nad besedilom ne izvedemo le ene od zgoraj naštetih operacij, ampak kar vse (to bomo demonstrirali tudi v naslednjem poglavju o klasifikaciji, ki za svoje delovanje potrebuje predprocesiranje s pomočjo vseh zgoraj naštetih operacij).

Podoben cevovod kot knjižnica SpaCy vsebuje tudi funkcija 'parse' iz knjižnice Pattern, ki prav tako združuje operacije lematizacije in oblikoskladenjskega označevanja, vendar ne vsebuje tokenizatorja (ali natančneje, njen vgrajeni tokenizator razdeli besedilo le na stavke). Knjižnica TextBlob pa povezuje operaciji tokenizacije in oblikoskladenjskega označevanja. Po drugi strani knjižnica NLTK ne vsebuje nikakršnega cevovoda ali kakega drugačnega smiselnega povezovanja različnih, pogosto skupaj uporabljenih operacij.

Sklenili smo preveriti, kako hitro so posamezne knjižnice sposobne izvesti kombinacijo treh operacij, tj. tokenizacije, lematizacije in oblikoskladenjskega označevanja. Meritve smo opravili na korpusu Gutenberg, in sicer na štirih različno dolgih delih tega korpusa (prvih 1000000 besed, prvih 500000 besed, prvih 250000 besed in prvih 125000 besed). V knjižnici NLTK smo uporabili privzeti tokenizator, Wordnet lematizator ter privzeti oblikoskladenjski označevalnik (gre za označevanje s pomočjo klasifikacije na podlagi maksimalne entropije), v knjižnici TextBlob pa smo uporabili kar njene privzete metode za te operacije (tokenizator in lematizator, ki smo ju testirali v prejšnjih poglavjih, in njen privzeti označevalnik iz knjižnice Pattern). Tudi pri teh meritvah hitrosti smo zaradi boljše preglednosti za čas uporabili logaritemsko skalo. Rezultati so predstavljeni na grafu (Slika 4.6).



Slika 4.6: Primerjava hitrosti kombinacije operacij tokenizacije, lematizacije in oblikoskladenjskega označevanja

Kot vidimo iz grafa, obstaja ogromna razlika v hitrosti med knjižnico NLTK in drugimi tremi knjižnicami, ki so kombinacijo nalog pri korpusu s 1000000 besedami opravile več kot desetkrat hitreje. K počasnosti knjižnice NLTK je verjetno odločilno prispeval ravno označevalnik. Druga zanimiva ugotovitev iz grafa je, da je najhitrejša knjižnica TextBlob, ki nima implementiranega cevovoda, združuje pa operaciji tokenizacije in označevanja. Očitno je njena implementacija posameznih metod dovolj hitra (njen tokenizator se je izkazal za najhitrejšega, prav tako njena implementacija oblikoskladenjskega označevalnika, lematizator pa je bil ravno tako med hitrejšimi), da je njihova izvedba prehitela celo SpaCyjev cevovod, ki je pristal na drugem mestu. Tudi knjižnica Pattern se je obnesla relativno dobro, za njen pristanek na tretjem mestu pa je verjetno kriva počasna implementacija lematizatorja v knjižnici 'parse', ki se je kot najpočasnejši izkazal že pri samostojnem testu hitrosti lematizacije.

Poglavje 5 Klasifikacija in analiza sentimenta

V tem poglavju bomo knjižnice med seboj primerjali po njihovih metodah za klasificiranje teksta. Klasifikacija besedil ni osnovna metoda pri procesiranju naravnega jezika, je pa ena najpogostejših operacij na besedilih, ki kot vhod zahteva že procesiran tekst. Tako imamo v tem poglavju priložnost, da uporabimo znanje iz prejšnjih poglavij in hkrati preverimo interoperabilnost posameznih knjižnic. Prav tako bo v tem poglavju poudarek na procesiranju slovenskega jezika

Klasifikacija teksta je postopek dodeljevanja pravilnih oznak (kategorij) delom teksta glede na rabo besed znotraj tega dela teksta. Primer klasifikacije je odločanje, ali je elektronska pošta 'spam' ali ne, določanje teme nekega članka v novicah ali pa odločanje, ali gre pri besedi 'klop' za nadležnega parazita ali objekt za sedenje. Obstaja več različnih vrst klasifikacije, v osnovi pa klasifikacijo delimo na nadzorovano (klasifikacija s pomočjo učne množice že klasificiranih primerov) in na nenadzorovano (iskanje vzorcev in povezav v nekategorizirani množici primerov), glede na število kategorij (binarni klasifikator na primer razdeli tekst na dve kategoriji, posamezen del teksta pa lahko pripada le eni kategoriji, ne pa obema hkrati) in glede na prekrivanje kategorij (multi-kategorijski klasifikator (*multi-label classifier*) dodeli tekstu eno ali več kategorij). Pri osnovnih klasifikacijskih opravilih so ponavadi vsi vhodni podatki obravnavani, kot da so med seboj neodvisni. Najpogostejša je nadzorovana klasifikacija, kjer klasifikator učimo na podlagi že pravilno kategoriziranega korpusa, vse kategorije pa so znane vnaprej.

Pri osnovni analizi sentimenta tekst klasificiramo glede na njegovo čustveno zaznamovanost, in sicer kot pozitivnega ali negativnega, običajno pa tem dvem razredom dodamo še srednji nevtralni razred za čustveno nezaznamovan tekst. Obstaja kar nekaj pristopov za določanje sentimenta, na grobo pa bi lahko te pristope razdelili na pristop s strojnim učenjem, leksikalni pristop in lingvistični pristop. Pri lingvističnem pristopu analiziramo slovnično strukturo besedila in na podlagi te strukture sklepamo na njegovo čustveno polarnost. Ta pristop se navadno izkaže za računsko zahtevnega in se zato uporablja le redko, saj ni primeren za uporabo v realnem času. Pri leksikalnih pristopih sentiment določamo na podlagi leksikona čustveno zaznamovanih besed. Tekst, ki vsebuje čustveno negativne besede iz leksikona, je klasificiran kot negativen, tekst, ki vsebuje čustveno pozitivne besede, pa je klasificiran kot

pozitiven. Prednost tega pristopa je predvsem njegova hitrost, problem pa je predvsem njegova nefleksibilnost (predvsem na internetu se jezikovna raba določenih besed s časom močno spreminja, prav tako se uporablja sleng in slovnično ne popolnoma pravilno besedišče, ki ga leksikon besed ne vsebuje) [5].

Pristop s strojnim učenjem deluje tako, da se klasifikator uči iz množice že kategoriziranih značilk (učna množica), da bi pozneje lahko označil še nekategorizirane množice značilk (nadzorovana klasifikacija). Ta pristop se zadnje čase uporablja najpogosteje, saj je primeren za analiziranje spletnih virov, kot so spletni komentarji in tviti. Za klasificiranje s pomočjo strojnega učenja obstaja več algoritmov z različnimi časovnimi zahtevnostmi, nekateri od njih pa so primerni za klasificiranje v realnem času. Problem tega pristopa je predvsem v izdelavi zadovoljive učne množice, kar je lahko precej zamudno opravilo.

Knjižnici NLTK in Pattern vsebujeta kar nekaj metod in algoritmov za klasificiranje teksta s pomočjo strojnega učenja, medtem ko knjižnici SpaCy in PyNLPl podpore za klasifikacijo teksta ne vsebujeta. Knjižnica TextBlob za klasificiranje uporablja klasifikatorje iz knjižnice NLTK, do katerih se dostopa s pomočjo različnih ovojnih razredov za vsak klasifikator posebej, ali pa kar direktno iz razreda 'TextBlob', ki vsebuje metodo 'classify'.

5.1 Izbira teksta

Pri izbiri teksta smo upoštevali dva kriterija. Kot prvo smo želeli slovenski tekst, saj smo želeli v praksi preveriti, kako se različne knjižnice obnesejo pri procesiranju slovenskega jezika. Označeni tekst je moral biti omejene dolžine, saj bomo na njem izvajali časovno precej zahtevne operacije. Za namen te diplomske naloge so se nam zato zdela najprimernejša kratka spletna besedila, kot so komentarji na spletnih straneh in komentarji na družbenih omrežjih. Od družbenih omrežij se nam je najprimernejše zdelo Twitter. To družbeno omrežje, ki je nastalo leta 2006, je kmalu pridobilo globalno popularnost in je imelo leta 2014 že 271 milijonov aktivnih uporabnikov. Twitter svojim uporabnikom dovoljuje objavo kratkih sporočil dolžine do 140 znakov, ki jih poznamo pod imenom 'tviti'. V povprečju je vsako sekundo objavljenih 6000 tvitov, kar pomeni 500 milijonov tvitov dnevno ali okoli 200 milijard tvitov v letu [5]. Twitter ni zanimiv le individualnim uporabnikom, ampak tudi korporacijam, saj je velika zakladnica osebnih informacij o njegovih uporabnikih, ki omogoča učinkovito spletno oglaševanje. Raziskave, ki so bile izvedene predvsem v angleškem jeziku, so že odkrile močno korelacijo med izmerjenim sentimentom do volilnih kandidatov v tvitih in rezultati volitev ter med izraženim sentimentom in borznimi gibanji [5], zato se nam je zdela analiza sentimenta v slovenskih tvitih zanimiva tema, saj je bilo na tem področju

opravljenih precej manj raziskav kot v angleščini, hkrati pa nas je presenetila precej slaba točnost doslej izvedenih pristopov v slovenščini [5].

Poleg tvitov smo se odločili tudi za analizo sentimenta v komentarjih na novičarskem portalu RTV Slovenija. Ti komentarji so po svojem jeziku, strukturi in dolžini zelo podobni tvitom in nam verjetno lahko ponudijo podobne informacije kot tviti. Tu sicer ni dolžinske omejitve 140 znakov, vendar pa le redki komentarji to dolžino dejansko presežejo.

Z Oddelka za prevajalstvo na inštitutu Jožefa Štefana nam je uspelo pridobiti množico 2000 komentarjev in 2000 tvitov na temo politike in športa [26], ki jim je bil ročno s pomočjo dveh ocenjevalcev določen sentiment (tviti in komentarji so bili kategorizirani v razrede '+', '-' in '0', ki predstavljajo pozitiven, negativen in nevtralen sentiment). To množico besedil smo uporabili kot temelj za generiranje učne in testne množice za algoritme strojnega učenja, še pred tem pa je bilo potrebno temeljito predprocesiranje.

5.2 Predprocesiranje teksta

Klasifikacijski algoritmi v knjižnici TextBlob in knjižnici Pattern sprejmejo vhod v obliki seznama dvojic (tekst, oznaka), knjižnica TextBlob pa kot vhod sprejme tudi csv ali json datoteko. Klasifikator (oziroma natančneje, v knjižnici TextBlob ovojni razredi za NLTK klasifikatorje, v knjižnici Pattern pa razred 'Document', ki tekst spremeni v 'vrečo besed') v teh knjižnicah nato sam poskrbi za tokenizacijo, lematizacijo in nadaljnjo obdelavo teksta. Takšen vhod je sicer prijazen do uporabnika, saj mu ni treba izvajati dodatnih operacij procesiranja naravnega jezika, a glede na to, da te knjižnice ne omogočajo lematizacije slovenskih tekstov, nam vhod takšne oblike ni ustrezal, saj smo želeli sami izvesti lematizacijo s pomočjo knjižnice Lemmagen. Knjižnica Pattern je dovolj fleksibilna, da njeni klasifikatorji kot vhod sprejemajo tudi tokeniziran tekst v obliki seznama, medtem ko to pri knjižnici TextBlob ni mogoče. Klasifikatorji knjižnice NLTK po drugi strani sprejmejo kot vhod le že močno obdelan tekst v obliki slovarja značilk (tako imenovana 'vreča besed' – več o tem bomo povedali v nadaljevanju).

Za začetek smo iz učne množice tvitov in komentarjev z Oddelka za prevajalstvo odstranili vse tvite in komentarje, pri katerih se ocenjevalca nista strinjala glede kategorije, ki sta jo dodelila. Tako smo iz prvotnih 4000 tvitov in komentarjev morali izločiti kar 823 primerov in na koncu pristali pri 3177 tvitih in komentarjih. Odstotek dvoumnih primerov, kjer se človeška ocenjevalca nista strinjala glede sentimenta, nas je presenetil ter navdal s pesimizmom glede strojnega reševanja problema dvoumnosti naravnega jezika in sarkazma, ki ga očitno nista znala najbolje rešiti tudi strokovno usposobljena človeška ocenjevalca.

Nato smo vse tvite in komentarje tokenizirali. Pri tem smo uporabili kar NLTK-jev privzeti tokenizator, ki je preprost za uporabo, a ne ravno hiter pri obdelavi velikih besedil, vendar pa hitrost pri tako majhnem številu tvitov ni bila vprašljiva. Ta tokenizator je tvite in komentarje razdelil na besede ter ločila predstavil kot posamezne žetone. Izločitev ločil iz tvitov in komentarjev pri empiričnih poskusih ni imela za posledico boljše točnosti klasifikacije, zato smo se odločili, da ločila pustimo. Nato smo vse pridobljene žetone lematizirali s pomočjo knjižnice Lemmagen. Naša funkcija za lematizacijo in tokenizacijo, ki kot vhod dobi seznam dvojic (tekst, oznaka), je bila videti takole:

```
def preprocess(list):

    tokenizedlist = []

    #tokenizacija in lematizacija
    lemmatizer = Lemmatizer(dictionary=lemmagen.DICTIONARY_SLOVENE)
    for text, tag in list:
        text = text.decode("utf8")

        #tokenizacija s pomočjo privzete NLTK funkcije za tokenizacijo
        tokens = nltk.word_tokenize(text)

        #lematizacija žetonov z lematizatorjem Lemmagen in  normalizacija
        #(sprememba vseh črk v male)
        lemmatizedText = [lemmatizer.lemmatize(x.lower()) for x in tokens]

        #žetone shranimo v seznam
        tokenizedlist.append((lemmatizedText, tag))
    return tokenizedlist
```

Zgornja funkcija vrne seznam dvojic oblike (tokeniziran in lematiziran tekst v obliki seznama, oznaka), ki je primeren za nadaljnje procesiranje.

5.3 Izračun značilk

Iskanje in uporaba relevantnih značilk in odločanje o tem, kako jih zakodirati, ima lahko odločilen vpliv na izgradnjo čim boljšega modela. Veliko časa pri izgradnji klasifikatorja se zato posveti temu delu. Čeprav je velikokrat možno, da dobimo solidne rezultate z uporabo dokaj preprostih značilk, se lahko točnost klasifikatorja drastično izboljša s pomočjo izgradnje zapletenih značilk, ki zahtevajo zelo dobro razumevanje samega klasifikacijskega problema. Značilke so ponavadi narejene na podlagi procesa poskusov in napak, ki ga vodi intuitivno razumevanje, katere informacije so relevantne za problem. Običajno obstaja meja glede števila značilk, ki jih uporabimo za klasifikacijo. Če je značilk preveč, se bo algoritem lahko preveč prilagodil učni množici in ne bo uspešen pri klasifikaciji novih problemov.

Ekstrakcija značilk je v našem primeru proces transformacije tvita ali komentarja (ki je tokeniziran in predstavljen v obliki seznama) v množico značilk, ki je primerna za uporabo pri klasificiranju. NLTK klasifikatorji zahtevajo množico značilk v obliki slovarja, zato je vedno potrebno vsak seznam besed spremeniti v Pythonov slovar. Ta slovar se imenuje »vreča besed« [24], saj besede nimajo določenega vrstnega reda, prav tako ni pomembno, kdaj in kolikokrat so se pojavile v seznamu. Klasifikatorji knjižnice Pattern kot argument sprejmejo slovar dvojic tekst (ki je tokeniziran ali pa ne), oznaka, vendar pa Patternov interni razred »Document« te dvojice interno prav tako spremeni v 'vrečo besed'. Funkcija, ki smo jo uporabili za spremembo tokeniziranega teksta v vrečo besed, je prikazana spodaj:

```
#funkcija za izdelavo značilk za klasifikatorje NLTK
def word_features(text):
    return dict([(word, True) for word in text])
```

Če vzamemo za primer vhoda tokeniziran tekst 'ta tekst je vreča besed', dobimo iz njega naslednjo 'vrečo besed':

{'ta': True, 'vreča': True, 'tekst': True , 'besed': True, 'je': True }

Tako v obeh knjižnicah kot značilke na koncu dobimo slovar dvojic žeton, 'boolean' (namesto 'booleanov' bi vsaj v NLTK lahko uporabili tudi kakšno numerično spremenljivko, kot je 'int' ali 'float')), kjer nam 'boolean' pove, ali posamezen tvit ali komentar vsebuje določen žeton. Vsak tvit in komentar sta tako spremenjena v vrečo v obliki slovarja, znotraj katere so besede, ki jih vsebuje.

5.4 Klasifikatorji

Knjižnici NLTK in Pattern vsebujeta več različnih klasifikatorjev. Klasifikacijo omogoča tudi knjižnica TextBlob, vendar pa uporablja klasifikatorje iz knjižnice NLTK. Knjižnici NLTK in Pattern vsebujeta naivni Bayesov klasifikator, knjižnica NLTK pa poleg tega vsebuje še klasifikator s pomočjo odločitvenega drevesa in klasifikator s pomočjo maksimalne entropije, katerega delovanje smo razložili že v poglavju o oblikoslovnih označbah. Knjižnica prav tako ponuja ovojni razred za dostop do klasifikatorjev v knjižnici Scikit-learn [29], zato smo v nadaljevanju preverili tudi točnost klasifikatorja s pomočjo naivnega Bayesa ter klasifikator s pomočjo metode podpornih vektorjev (support vector machine, v nadaljevanju SVM) iz te knjižnice. Knjižnica Pattern vsebuje še SVM klasifikator, klasifikator s pomočjo k-najbližjih sosedov (k-nearest neighbours, v nadaljevanju KNN) ter enonivojski perceptron (single layer perceptron, v nadaljevanju SLP).

5.4.1 Naivni Bayesov klasifikator

Naivni Bayesov klasifikator kategorizira po naslednji formuli:

$$P(oznaka \mid značilka) = \frac{P(oznaka) * P(značilka \mid oznaka)}{P(značilka)} \tag{5.1}$$

P(oznaka) je vnaprejšnja verjetnost, da se oznaka pojavi, torej koliko značilk od vseh ima to oznako v učni množici. P(značilka | oznaka) je vnaprejšnja a priori verjetnost, da ima značilka določeno oznako. P(značilka) je verjetnost, da se ta značilka pojavi (na primer, če se v učni množici ta značilka pojavi dvakrat in gre za množico s stotimi primeri, je verjetnost 2%). P(oznaka | značilka) je verjetnost, ki jo iščemo, torej verjetnost, da ima določena značilka v resnici to oznako. Če je verjetnost velika, potem smo lahko skoraj prepričani, da je klasifikacija pravilna. Verjetnost določene oznake za celoten tekst dobimo tako, da med seboj zmnožimo verjetnosti vseh značilk (verjetnosti, da ima določena značilka določeno oznako), ki jih tekst ima. Na koncu le še izberemo oznako z največjo verjetnostjo [24].

Prednost naivnega Bayesa je, da pri njem točno vemo, kakšen je postopek ocenjevanja. Tako ima ta klasifikator v NLTK tudi metode, ki nam omogočajo, da bolje razumemo vrednost vhodnih podatkov. Metoda 'show_most_informative_feature' nam pove, katere posamezne značilke so najboljše za klasifikacijo, kar je zelo uporabna povratna informacija, ki nam pomaga bolje razumeti značilnosti naših podatkov. Vendar ima ta klasifikator tudi pomanjkljivosti, in sicer je problematična predvsem njegova 'naivnost', ki predpostavlja, da so vse značilke med seboj neodvisne. V resničnem svetu ta predpostavka preprosto ne drži. Problem nastopi, če imamo dve med seboj močno povezani značilki. Klasifikator v tem primeru upošteva doprinos obeh značilk in na ta način 'neupravičeno' podvoji moč argumenta za določeno oznako.

5.4.2 Odločitvena drevesa

Klasifikacija s pomočjo odločitvenega drevesa deluje tako, da se naredi drevesna struktura, kjer je vsako križišče ime značilke, veje pa so lastnosti značilke. Če sledimo vejam vse do konca, pridemo do listov, ki so oznake (razredi). Klasifikator gradi drevo od spodaj navzgor, na koncu pa drevo preuredi tako, da je pot do končne odločitve pri vseh značilkah čim krajša in da so na vrhu značilke, ki so najbolj informativne. Prednost odločitvenega drevesa je podobna kot pri naivnem Bayesu, in sicer da točno vemo, na podlagi kakšnih pravil klasifikator klasificira, kar nam spet da pomembno informacijo o naših podatkih [24].

5.4.3 Klasificiranje z maksimalno entropijo

Klasificiranje s pomočjo maksimalne entropije poteka tako, da klasifikator zakodira označene značilke v vektorje, ti vektorji pa so pozneje uporabljeni za izračun uteži za vsako značilko, ki so nato kombinirane z namenom, da se določeni množici značilk pripiše pravilna oznaka. Glavna ideja pri tej klasifikaciji je, da se zgradi model, ki vsebuje več verjetnostnih distribucij, ki ustrezajo podatkom, na koncu pa se izbere distribucijo z največjo entropijo [24].

5.4.4 Metoda podpornih vektorjev (SVM)

SVM klasifikator temelji na reprezentaciji posameznih besedil v večdimenzionalnem matematičnem prostoru, dimenzije pa so med seboj ločene s pomočjo hiperravnin. Ta klasifikator se je na splošno obnesel dobro, prav tako njegovo delovanje ni prepočasno, problem pa je, da deluje kot črna skrinjica, tako da nimamo vpogleda v njegovo delovanje in ne moramo pridobiti kriterijev, na podlagi katerih klasificira, ter s tem informacij o naši učni množici [10].

5.4.5 Metoda najbližjih sosedov (KNN)

KNN klasifikator klasificira tekst na podlagi določenega števila (to število določa 'k') njemu najbolj podobnih tekstov iz učne množice, podobnost pa računa na podlagi določene funkcije (na primer kosinusna podobnost). Ta klasifikator je precej časovno zahteven, vendar pa ima tako kot naivni Bayesov klasifikator prednost, da točno vemo, kaj se pri klasifikaciji dogaja [10].

5.4.6 Enonivojski perceptron (SLP)

SLP klasifikator je preprosta nevronska mreža, ki temelji na oteženih povezavah med nevroni, katerih uteži se iterativno prilagajajo učni množici med treningom. Pri tem klasifikatorju imamo podoben problem kot pri klasifikatorju SVM, tj. da ne vemo točno, kaj se dogaja med samo klasifikacijo [10].

5.4.7 Uporaba klasifikacije v knjižnici NLTK

V knjižnici NLTK vsi klasifikatorji dedujejo po razredu 'ClassifierI', zato so metode za njihovo učenje, testiranje in klasificiranje pri vseh klasifikatorjih enake. Kot primer si poglejmo kodo za učenje, testiranje in klasificiranje klasifikatorja s pomočjo naivnega Bayesa:

```
import nltk
#učenje klasifikatorja
classifier =nltk.NaiveBayesClassifier.train(training_set)

#testiranje klasifikatorja
accuracy = nltk.classify.util.accuracy(classifier, test_set)

#klasificiranje tokeniziranega tvita
classifier.classify(word_features(['Danes','biti', 'lep', 'dan']))
```

Za izračun preciznosti, priklica in F-mere (več o teh merah bomo povedali v naslednjem poglavju), ki smo jih uporabili za merjenje uspešnosti klasifikatorjev, je potrebnega malo več dela, kodo pa prilagamo v dodatku diplomske naloge. Tudi v knjižnici Pattern imajo vsi klasifikatorji enake metode za učenje, testiranje in klasificiranje. Kot primer si poglejmo kodo za učenje, testiranje in klasificiranje klasifikatorja SVM:

```
from pattern.vector import SVM

#učenje klasifikatorja
classifier = SVM(train=trainset)

#testiranje klasifikatorja
accuracy, precision, recall, f1 = classifier.test(testset)

#klasificiranje tokeniziranega tvita
classifier.classify('Danes','biti', 'lep', 'dan'])
```

Funkcija 'test' vrne točnost, preciznost, priklic in F-mero, vendar pa naj omenimo, da gre tu za drugačno točnost, kot bi jo želeli imeti mi, zato smo točnost merili s pomočjo lastne funkcije. Več o tem bomo povedali v naslednjem poglavju.

5.5 Kriteriji uspešnosti klasifikacije in primerjave klasifikatorjev

Vse zgoraj naštete klasifikatorje smo primerjali po točnosti. Zaradi majhnosti učne množice smo uporabili 10-kratno prečno preverjanje, in sicer smo desetkrat iz celotne množice tvitov in komentarjev izločili drugo testno množico, ki je vsebovala desetino vseh tvitov in komentarjev, ter klasifikator nato učili na ostalih tvitih in komentarjih v množici. Naučen klasifikator smo desetkrat testirali na izločeni desetini množice ter na koncu izračunali povprečje meritev. Kriteriji uspešnosti klasifikacije so bili trije. Točnost (*accuracy*) smo računali po formuli:

$$točnost = \frac{število\ pravilno\ klasificiranih\ primerov}{število\ vseh\ primerov} \tag{5.2}$$

Poleg točnosti pri klasifikaciji pogosto uporabljamo še druge mere za ocenjevanje uspešnosti klasifikatorjev. Najpogostejša sta preciznost (precision) in priklic (recall). Nepravilna pozitivna klasifikacija se zgodi, ko klasifikator klasificira množico značilk z oznako, ki je ne bi smela imeti. Nepravilna negativna klasifikacija pa se zgodi, ko klasifikator ne dodeli oznake množici značilk, ki bi morala imeti to oznako. Pri binarni klasifikaciji se ti dve napaki zgodita hkrati. Preciznost je pomanjkanje nepravilnih pozitivnih klasifikacij, priklic pa je pomanjkanje nepravilnih negativnih klasifikacij. Tako velja, da če ima klasifikator nizko preciznost, so negativni primeri pogosto klasificirani kot pozitivni. Če pa ima klasifikator nizek priklic, to pomeni, da klasifikator zgreši mnogo primerov, ki bi jih moral klasificirati za pozitivne. Zato sta ti dve meri pogosto v obratnem sorazmerju: boljši ko je priklic, manjša je preciznost in obratno. Hkrati pa nam te mere dajo boljšo sliko o dejanski uspešnosti klasifikatorja kot točnost [10]. Za boljšo predstavo si poglejmo matriko klasifikacij (Slika 5.1):

Razred	**Napovedani razred**	
	Pravilen	Napačen
Pravilen	Pravilna pozitivna klasifikacija (TP – true positive)	Nepravilna negativna klasifikacija (FN – false negative)
Napačen	Nepravilna pozitivna klasifikacija (FP – false positive)	Pravilna negativna klasifikacija (TN – true negative)

Slika 5.1: Konfuzijska matrika

Preciznost se računa po sledeči formuli:

$$preciznost = \frac{TP}{(TP + FP)} \tag{5.3}$$

Priklic se računa po sledeči formuli:

$$priklic = \frac{TP}{(TP + FN)} \tag{5.4}$$

Glede na to, da v našem primeru ni šlo za binarno klasificiranje, ampak smo imeli klasifikacijo v tri razrede, smo preciznost in priklic izračunali za vsak razred posebej ter nato izračunali povprečje priklicev in preciznosti, ki smo jih dobili za vsak razred.

Poleg zgoraj naštetih kriterijev smo za merjenje uspešnosti klasifikatorjev uporabili še F-mero, ki je definirana kot utežena harmonična sredina preciznosti in priklica in se izračuna po spodnji formuli:

$$F - mera = \frac{1}{\frac{alpha}{n} + \frac{1 - alpha}{p}} \tag{5.5}$$

kjer sta n = preciznost in p = priklic, alpha pa je konstanta, ki je privzeto 0.5.

5.5.1 Meritve

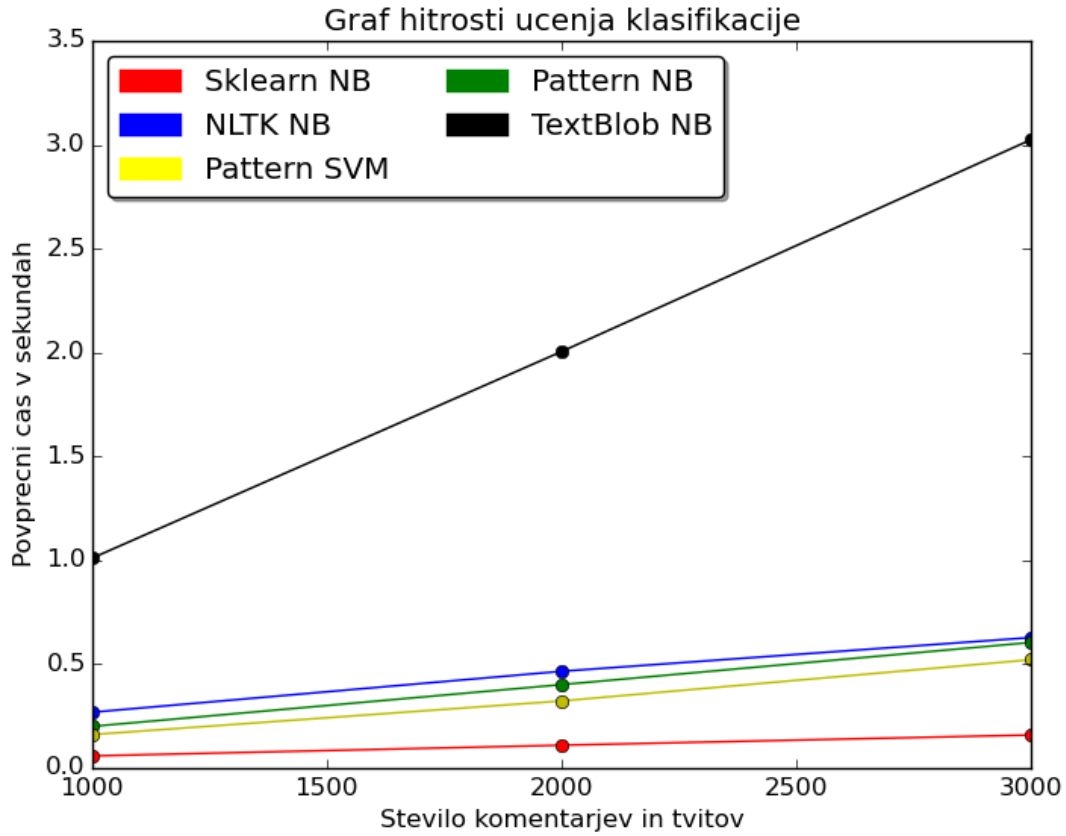
	Točnost	**Preciznost**	**Priklic**	**F-mera**
Naivni Bayes (NLTK)	0,669	0,693	0,636	0,663
Maksimalna Entropija	0,607	0,691	0,544	0,609
Odločitveno drevo	0,613	0,626	0,575	0,600
SVM (Scikit-learn)	0,663	0,658	0,664	0,661
Naivni Bayes (Scikit-learn)	0,670	0,686	0,643	0,664
SLP	0,595	0,598	0,594	0,596
Naivni Bayes (Pattern)	0,641	0,642	0,627	0,634
SVM (Pattern)	0,652	0,682	0,623	0,651
KNN	0,593	0,585	0,586	0,586
Večinski klasifikator	0,430	/	/	/

Tabela 5.1: Uspešnost klasifikatorjev

Meritve uspešnosti klasifikacije za vse zgoraj naštete klasifikatorje so predstavljene v tabeli (Tabela 5.1). Najboljši rezultati v tabeli so označeni z rumeno. Za začetek naj povemo, da so vsi klasifikatorji boljši od večinskega klasifikatorja (klasifikator, ki vsem tvitom in komentarjem pripiše kategorijo večinskega razreda). V množici 3177 tvitov in komentarjev je bilo 741 pozitivnih tvitov, 1360 negativnih tvitov in 1076 nevtralnih tvitov, kar pomeni, da je večinski razred negativen, večinski klasifikator pa je dosegel točnost 43%. Najboljše se je tako po točnosti (0,670) kot F-meri (0,664) odrezal klasifikator s pomočjo naivnega Bayesa iz knjižnice Scikit-learn. Zelo tesno mu sledi klasifikator s pomočjo naivnega Bayesa iz knjižnice NLTK, ki le za stotinko zaostaja po obeh zgoraj omenjenih kriterijih, kar pomeni, da sta ta dva klasifikatorja pravzaprav enakovredna. Razlika med njima je morda le v tem, da klasifikator iz knjižnice NLTK dosega boljšo preciznost, klasifikator iz knjižnice Scikit-learn pa boljši priklic. Dobro sta se odrezala tudi klasifikatorja SVM iz knjižnic Scikit-learn in Pattern s točnostjo 0,663 oziroma 0,652 in F-mero 0,661 oziroma 0,651. Najslabše sta se odrezala KNN in SLP klasifikator, ki jima ni uspelo preseči praga 0,6 po nobenem kriteriju uspešnosti. Malce bolje, a ne posebej dobro sta se odrezala tudi klasifikator s pomočjo maksimalne entropije (ki ima zelo dobro preciznost, a zelo slab priklic) ter odločitveno drevo.

5.6 Meritve hitrosti učenja klasifikatorjev

Zaradi velike množice različnih klasifikatorjev smo se odločili, da po hitrosti primerjamo le štiri po uspešnosti najboljše klasifikatorje: klasifikatorje s pomočjo naivnega Bayesa iz knjižnic NLTK, Scikit-learn ter Pattern, in pa klasifikator SVM iz knjižnice Pattern. Izmerili smo tudi hitrost TextBlob implementacije klasifikatorja s pomočjo naivnega Bayesa iz knjižnice NLTK, saj sama implementacija lahko vpliva na hitrost. Merili smo čas učenja posameznih klasifikatorjev, in sicer smo izmerili hitrost učenja na množici 1000, 2000 in 3000 tvitov. Kot že pri prejšnjih meritvah hitrosti smo vsako meritev izvedli desetkrat in nato kot rezultat predstavili povprečni čas. Meritve so predstavljene na grafu (Slika 5.2).



Slika 5.2: Meritve hitrosti učenja klasifikatorjev

Za najhitrejšega se je izkazal klasifikator iz knjižnice Scikit-learn, ki je z množico 3000 tvitov opravil v manj kot 0,2 sekunde. Drugi, a vendar dvakrat počasnejši je bil klasifikator SVM iz knjižnice Pattern, klasifikatorja s pomočjo naivnega Bayesa iz knjižnic NLTK in Pattern pa sta za množico 3000 tvitov porabila več kot trikrat več časa kot klasifikator s pomočjo naivnega Bayesa iz knjižnice Scikit-learn. Daleč najpočasnejša je bila implementacija NLTK klasifikatorja s pomočjo naivnega Bayesa v knjižnici TextBlob, ki je bil pri največjem korpusu kar petnajstkrat počasnejši od najhitrejšega klasifikatorja. Očitno so implementacije istih tipov klasifikatorjev v različnih knjižnicah razlikujejo. Standardni odkloni so se pri vseh meritvah gibali okoli dveh stotink sekunde.

Potrebno je omeniti, da na hitrost klasifikatorjev močno vpliva število značilk. Hitrost klasifikatorja iz knjižnice TextBlob nam je uspelo izmeriti šele potem, ko smo mu kot argument dodali funkcijo za izračun značilk, ki računa značilke na že prej opisan način, ki smo ga uporabili tudi v knjižnici NLTK. Medtem ko smo mi po omenjenem postopku za izračun značilk za vsak tvit dobili v povprečju okoli 30 značilk, pa TextBlobov privzeti algoritem za izračun značilk značilke računa drugače. Algoritem najprej naredi seznam vseh

besed iz učne množice, nato pa za vsak tvit naredi slovar značilk, ki vsebuje vse besede v učni množici (značilka oblike 'beseda:True' pomeni, da tvit vsebuje besedo, značilka oblike 'beseda: False' pa pomeni, da tvit besede ne vsebuje). Naša učna množica je vsebovala okoli 77000 besed, od tega je bilo okoli 10000 besed unikatnih. Če imamo 3 množice s po 1000, 2000 in 3000 tvitov, to pomeni, da klasifikator samo za sprehod čez vse značilke potrebuje 60 milijonov korakov, prav tako smo meritve za vsako množico izvedli desetkrat, kar pomeni že 600 milijonov korakov, medtem ko so ostali algoritmi za sprehod potrebovali le okoli 1800000 korakov, kar je okoli 333-krat manj. TextBlobov klasifikator s privzetim algoritmom za izračun značilk je zato po našem mnenju primeren le za učenje na zelo majhnih učnih množicah, saj je na vseh ostalih preprosto prepočasen.

5.7 Izboljšave uspešnosti klasifikacije

V tem poglavju želimo preveriti, ali lahko s pomočjo raznih tehnik za procesiranje naravnega jezika izboljšamo uspešnost klasifikacije. Preizkusili bomo različne metode iz različnih knjižnic in na ta način preverili tudi interoperabilnost posameznih knjižnic. Do sedaj najboljšo uspešnost je dosegel klasifikator s pomočjo naivnega Bayesa iz knjižnice Scikit-learn, ki je dosegel 67% točnost, 68,8% preciznost, 64,3% priklic in 66,4% F-mero. Ta uspešnost bo mejnik, ki ga v tem poglavju želimo preseči.

5.7.1 Filtriranje s pomočjo oblikoskladenjskega označevanja

Uspešnost klasifikacije najboljšega preizkušenega klasifikatorja (klasifikatorja s pomočjo naivnega Bayesa iz knjižnice Scikit-learn) smo najprej želeli izboljšati s pomočjo filtriranja tvitov in komentarjev na podlagi oblikoskladenjskega označevanja.

Za začetek je bilo treba naučiti oblikoskladenjski označevalnik na slovenskem jeziku. Za učno množico smo vzeli slovenski korpus ssj500k [27], ki vsebuje 500 tisoč ročno označenih besed iz slovenskega jezika. Korpus je nastal v okviru projekta JOS [32] in je sestavljen iz korpusa jos100k ter dodatnih 400.000 besed iz enomilijonskega korpusa jos1M [22]. Dosegljiv je v XML formatu, poleg oblikoslovnih označb pa vsebuje tudi leme vseh besed. Oblikoslovne označbe imajo obliko msd kod [20], ki so sestavljene iz več znakov. Prvi znak pove, za kakšno besedno vrsto gre, in sicer obstaja 11 možnosti: S pomeni samostalnik, G glagol, P pridevnik, R prislov, D predlog, V veznik, L členek, M medmet, K številnik, O okrajšava in Z zaimek. Naslednji znaki določajo spol, sklon, število, čas, osebo in druge lastnosti besede v korpusu. Za naše potrebe se nam je zdelo določanje zvrsti posamezne besede dovolj, zato smo upoštevali le prvi znak msd kode.

S pomočjo Pythonovega razčlenjevalnika za XML dokumente 'elementtree' [6] smo se sprehodili čez celoten dokument in iz njega izločili vse leme besed (ker bo naš označevalnik kot argument dobil že lematizirane tvite in komentarje, nismo potrebovali besed, ampak le njihove leme) ter prve znake msd kode. Žetonom v korpusu brez msd kode (ločila ter drugi žetoni) smo namesto prve črke msd oznake kot par v dvojici pripisali kar samega sebe. Naša implementacija zgornjih operacij je bila videti takole:

```
import xml.etree.ElementTree as ET

#funkcija za branje korpusa ssj500k v XML obliki
def createSloveneTestSet():
    tree = ET.parse('ssj500kv1_3.xml')
    root = tree.getroot()
    text = []
    for child in root[1][0]:
        for grandchild in child:

            #poiščimo vse stavke v korpusu
            for t in grandchild.findall('{http://www.tei-c.org/ns/1.0}s'):
                sentence = []

                #vsi žetoni v stavku
                for w in t:
                    attributes = w.attrib

                    #če imajo lemo, shranimo lemo in msd oznako
                    if 'lemma' in attributes.keys():
                        sentence.append((attributes['lemma'],
                            attributes['msd'][0]))

                    #če nimajo leme (ne gre za besedo), shranimo dvojico
                    #(žeton, žeton)
                    elif w.text:
                        sentence.append((w.text, w.text))
                text.append(sentence)
    return text
```

Kot rezultat funkcije smo dobili seznam stavkov naslednje oblike, ki je bil primeren za učenje našega označevalnika:

[[(Hiša, S), (biti, G), (zelena, P),(., .)], [(Avto, S), (peljati, G), (po, D), (cesta, S), (., .)]...]

5.7.1.1 Izbira označevalnika in meritve

Pri odločitvi za označevalnik smo se zanašali na meritve iz poglavja o oblikoskladenjskem označevanju, kjer se je zelo dobro odrezal NLTK-jev označevalnik s pomočjo klasifikacije, ki za klasifikacijo uporablja algoritem naivnega Bayesa. Boljšo točnost od njega je dosegel le označevalnik iz knjižnice SpaCy, ki pa ga na žalost ni mogoče učiti na novi učni množici. Za začetek smo označevalnik naučili s pomočjo 9/10 korpusa ssj500k in njegovo točnost testirali

na preostali zadnji desetini korpusa. Točnost je dosegla 98%. Označevalnik smo nato v nadaljevanju naučili na celotnem korpusu ssj500k ter ga uporabili za oblikoskladenjsko označevanje naših tvitov in komentarjev.

Ko smo tako dobili označbe za vse besede in druge žetone v naših tvitih in komentarjih, smo v okviru predprocesiranja iz tvitov in komentarjev izločali besede z določenimi oblikoskladenjskimi označbami in nato pri določanju sentimenta tvitov upoštevali besede z določeno oznako. Dobili smo naslednje rezultate (Tabela 5.2):

Vsebovane vrste besed	**Preciznost**	**Priklic**	**F-mera**	**Točnost**
S,G,P	0,612	0,574	0,592	0,602
S,G,Z,P,K,D,V,R	0,664	0,598	0,629	0,634
S,G,Z,P,K,D,V,R,M,L,O	0,673	0,610	0,640	0,644

Tabela 5.2: Rezultati meritev pri liberalnem filtriranju s pomočjo oblikoskladenjskega označevanja

Kot vidimo iz zgornje tabele, smo uspešnost klasifikatorja s tem precej liberalnim filtriranjem samo poslabšali. Najslabše se je obnesel filter, ki je v tvitih in komentarjih pustil le samostalnike, glagole in pridevnike (to so ponavadi besede z največjo sentimentalno vrednostjo [24]), saj je dosegel le 60,2% točnost in 59,2% F-mero. Najbolje se je obnesel filter, ki je v tvitih in komentarjih pustil vse besede s pripisanimi oblikoskladenjskimi oznakami (izločil je le ločila in besede, ki jim označevalnik ni znal pripisati oznak), ki je dosegel 64,4% točnost in 64% F-mero, a tudi ta je uspešnost klasifikatorja le poslabšal.

Po neuspehu našega liberalnega filtriranja smo se odločili za konzervativnejši pristop, kjer smo namesto, da bi v tvitih in komentarjih pustili besede z določenimi oblikoskladenjskimi oznakami, raje odstranili le določene besede z oblikoskladenjskimi oznakami. Na ta način smo v tvitih in komentarjih pustili ločila, osebna imena in druge neoznačene žetone, ki očitno pozitivno vplivajo na klasifikacijsko uspešnost. Dobili smo naslednje rezultate (Tabela 5.3):

Izločene vrste besed	Preciznost	Priklic	F-mera	Točnost
Vezniki	0,680	0,642	0,661	0,668
Predlogi	0,680	0,643	0,661	0,667
Števniki	0,686	0,646	0,665	0,672
Členki	0,686	0,646	0,666	0,673
Medmeti	0,687	0,641	0,663	0,669
Okrajšave	0,688	0,646	0,666	0,673
Členki in okrajšave	0,684	0,646	0,664	0,673
Členki ,okrajšave in števniki	0,685	0,649	0,667	0,674

Tabela 5.3: Rezultati meritev uspešnosti klasifikacije pri konzervativnem filtriranju s pomočjo oblikoskladenjskega označevanja

V zgornji tabeli lahko opazimo, da se točnost, preciznost in priklic rahlo povečajo, če iz tvitov in komentarjev odstranimo ali števnike ali členke ali okrajšave, in sicer na uspešnost klasifikacije najbolje vpliva odstranitev medmetov ali členkov, saj na ta način dobimo klasifikator s 67,3% točnostjo in 66,6% F-mero, kar je izboljšava za 0,3% pri točnosti in 0,2% pri F-meri. Če hkrati odstranimo členke, okrajšave in števnike, pa dobimo 67,4% točnost ter 66,7% F-mero.

Čeprav velja, da lahko s previdnim filtriranjem na podlagi oblikoskladenjskih označb izboljšamo uspešnost klasifikacije, pa lahko na podlagi naših rezultatov zaključimo, da ta doprinos ni posebej velik. Tako velja, da oblikoskladenjsko označevanje pri analizi sentimenta ni nujno potrebna operacija, lahko pa doda klasifikatorju nekaj dodatne ostrine, vendar le z zelo previdno uporabo.

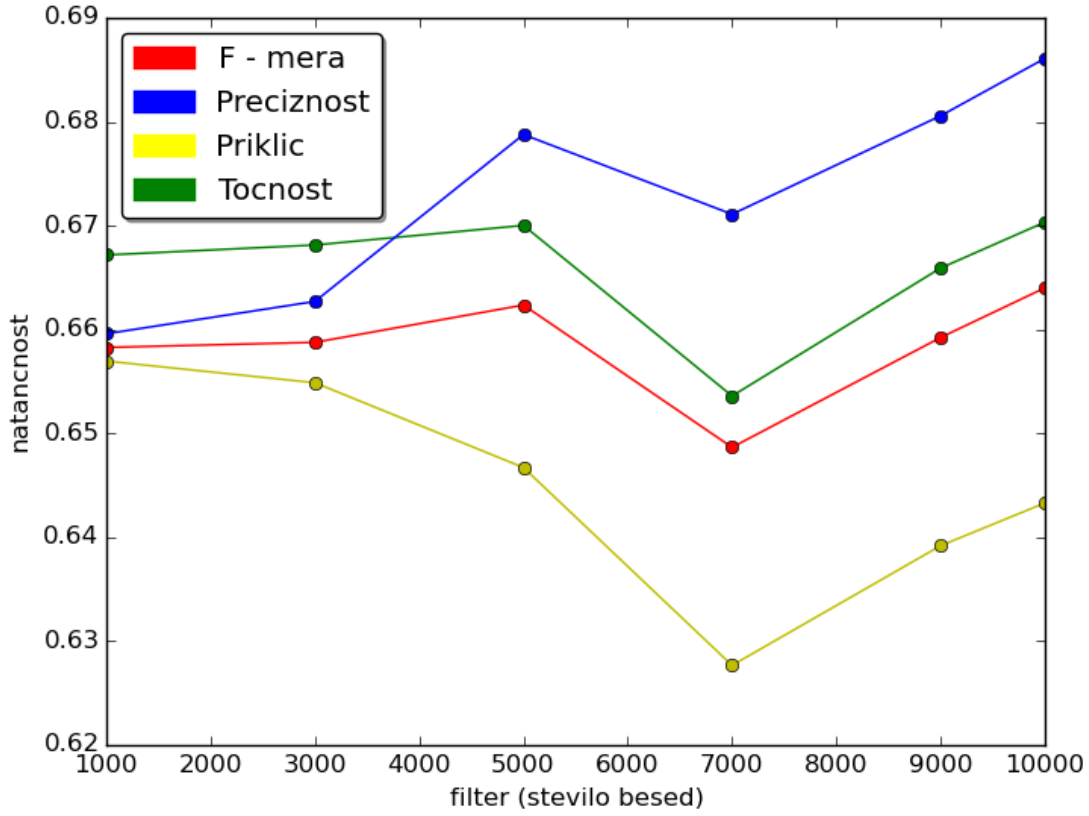
5.7.2 Filtriranje s pomočjo TF-IDF

Funkcija Pattern vključuje razred 'Model', ki omogoča filtriranje na podlagi različnih statističnih kriterijev, privzeto pa za filtriranje uporabi TF-IDF (term frequency – inverse

document frequency). Njegova funkcija 'feature_selection' izračuna TF-IDF za vsako besedo v učni množici in nato vrne n besed z največjo izračunano vrednostjo. Poglejmo si, kako smo implementirali ta postopek v naši programski kodi:

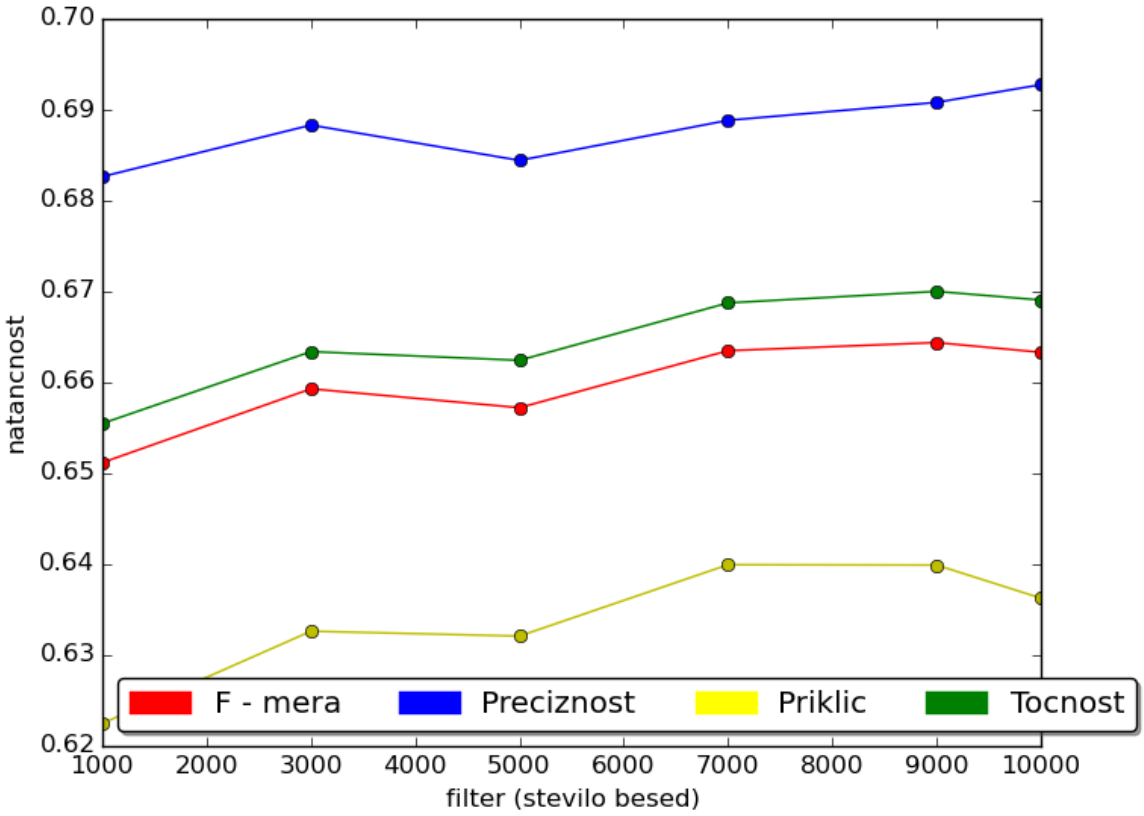
```
from pattern.vector import Model, Document

#funkcija za izdelavo značilk za klasifikatorje NLTK
def word_features(text, importantwords):

    #iz seznama besed naredimo slovar značilk, v slovar vključimo le
    #besede, ki so v množici po TF-IDF najboljše ocenjenih besed
    #('importantwords')
    return dict([(word, True) for word in text if word in importantwords])

#funkcija za izdelavo filtriranih seznamov dvojic tekst, kategorija za
#klasifikatorje knjižnice Pattern
def filterFeatures(trainset, importantwords):
    l = []

    #sprehod čez učno množico, v vsakem tvitu in komentarju za vsako besedo
    #pogledamo, če beseda je v množici po TF-IDF najbolje ocenjenih besed,
    #in če je, to besedo zapišemo v nov seznam, ki ga vrnemo kot izhod
    for text, category in trainset:
        l.append(([word for word in text if word in importantwords],
                  category))
    return l


#dvojice (tekst, oznaka) iz naše učne množice spremenimo v vrečo besed s
#pomočjo razreda 'Document'
data = [Document(text, type=category, stopwords=False) for text, category
        in training_set]

#iz dokumentov naredimo model
model = Model(documents=data)

#filtriranje modela s pomočjo TF-IDF, argument n pove, koliko besed z
#največjo vrednostjo TF-IDF pustimo v modelu.
importantwords = set(model.feature_selection(top=n))

#izdelava značilk za tvite in komentarje v učni množici za klasifikatorje
#knjižnice NLTK
training_set = [(word_features(text, importantwords), category) for text,
                category in training_set]

#izdelava značilk za tvite in komentarje v testni množici za klasifikatorje
#knjižnice NLTK
test_set =  [(word_features(text, importantwords), category) for text,
             category in test_set]

#izdelava filtriranih seznamov za klasifikatorje v knjižnici Pattern
trainset = filterFeatures(trainset, importantwords)
testset = filterFeatures(testset, importantwords)
```

Na koncu tako dobimo učno in testno množico tvitov in komentarjev, ki vsebujejo le n besed, ki so bile najbolje ocenjene glede na TF-IDF. Naša učna množica je vsebovala okoli 77000 besed (dolžina se je malenkostno spreminjala zaradi desetkratnega prečnega preverjanja, znotraj katerega je desetkrat generirana drugačna učna množica), od tega je bilo približno 10000 besed unikatnih. Tako smo se odločili, da meritve uspešnosti klasifikacije izvedemo pri filtrih 1000, 3000, 5000, 7000 in 9000 po TF-IDF najbolje ocenjenih besed. Meritve smo najprej izvedli za klasifikator s pomočjo naivnega Bayesa iz knjižnice Scikit-learn, ki se je po uspešnosti klasifikacije izkazal za najboljšega.



Slika 5.3: Meritve filtriranja s pomočjo TF-IDF za klasifikator s pomočjo naivnega Bayesa iz knjižnice Scikit-learn

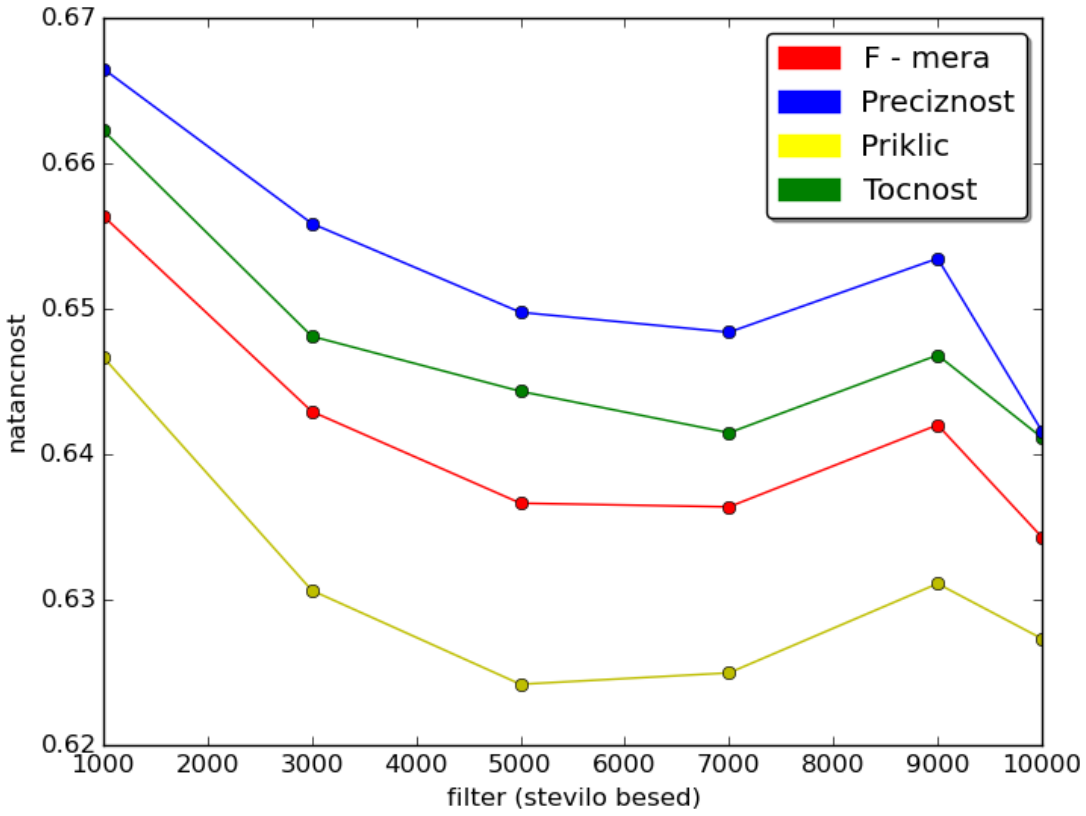
Iz grafa (Slika 5.3) je razvidno, da filtriranje ne izboljša uspešnosti tega klasifikatorja, ampak jo celo poslabša. Klasifikator najboljšo uspešnost dosega pri 10000 besedah, kjer gre za nefiltrirano učno množico, ki vsebuje približno 10000 unikatnih besed, najslabšo pa pri filtru 7000 besed. Poglejmo si še rezultate za klasifikator s pomočjo naivnega Bayesa iz knjižnice NLTK (Slika 5.4).



Slika 5.4: Meritve filtriranja s pomočjo TF-IDF za klasifikator s pomočjo naivnega Bayesa iz knjižnice NLTK

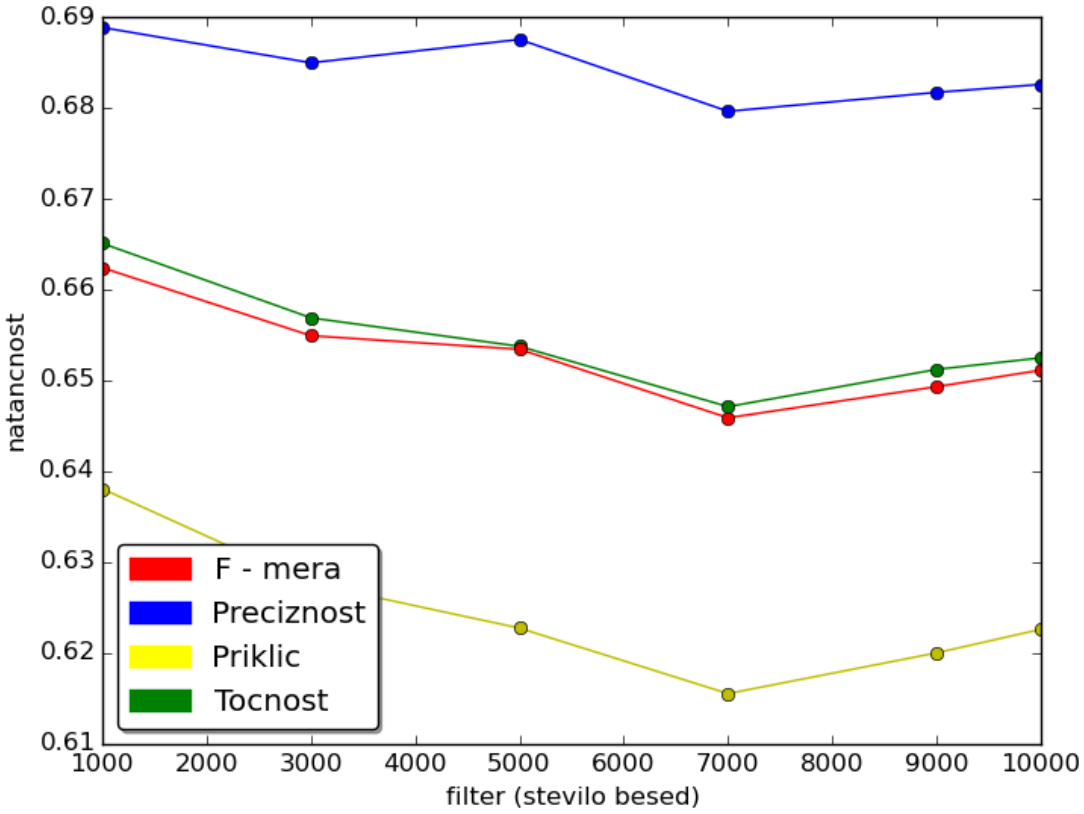
Tu lahko opazimo malenkostno izboljšanje uspešnosti klasifikacije pri filtru 9000 besed, ki pa je skoraj neopazno. Najslabšo uspešnost klasifikator dosega pri najmanjšem filtru, torej pri 1000 besedah.

Ker smo opazili, da filtri očitno različno vplivajo na različne implementacije klasifikatorja s pomočjo naivnega Bayesa, smo pogledali še vpliv filtriranja na klasifikator s pomočjo naivnega Bayesa iz knjižnice Pattern (Slika 5.5):



Slika 5.5: Meritve filtriranja s pomočjo TF-IDF za klasifikator s pomočjo naivnega Bayesa iz knjižnice Pattern

Pri tem klasifikatorju filter 1000 besed dejansko pozitivno vpliva na uspešnost, in sicer točnost in F-mero izboljša za okoli 2 odstotka. Očitno so implementacije klasifikatorja s pomočjo naivnega Bayesa v treh knjižnicah različne, in sicer so nekateri klasifikatorji bolj odporni na prilagajanje učni množici zaradi prevelikega števila značilk. Za primerjavo si poglejmo še, kako filtriranje vpliva na SVM klasifikator (Slika 5.6).



Slika 5.6: Meritve filtriranja s pomočjo TF-IDF za SVM klasifikator iz knjižnice Pattern

Pri tem klasifikatorju ima filtriranje manjši vpliv kot pri klasifikatorjih s pomočjo naivnega Bayesa. Največjo uspešnost po vseh kriterijih klasifikator dosega pri filtru 1000 besed.

Če povzamemo vse meritve, lahko rečemo, da filtriranje ne vpliva bistveno na uspešnost klasifikatorjev, vplivi pa se razlikujejo od klasifikatorja do klasifikatorja. Prav tako nam na žalost s pomočjo nobenega klasifikatorja ni uspelo izboljšati najboljše izmerjene točnosti, ki jo je dosegel klasifikator s pomočjo naivnega Bayesa iz knjižnice Scikit-learn.

5.7.3 Drugi poskusi izboljšave uspešnosti klasifikacije

Če smo pri prejšnjih poskusih izboljšav uspešnosti klasifikacije to želeli doseči s pomočjo filtriranja učne množice, pa smo se tu odločili, da poskusimo boljšo uspešnost doseči z razširitvijo naše učne množice. Knjižnica NLTK ima par metod za iskanje kolokacij (pogostih besednih zvez), ki smo jih želeli preizkusiti v praksi in hkrati preveriti, če lahko z vključitvijo teh besednih zvez v našo učno množico dosežemo boljšo uspešnost klasifikacije. Tako smo v naši učni množici poiskali 100 po kriteriju PMI (*Pointwise mutual information*) najbolje

ocenjenih kolokacij, ki se vsaj trikrat pojavijo v naši učni množici, in jih vključili v množico kot dodatne značilke. Kolokacije smo poiskali s pomočjo spodnje kode:

```
import nltk
from nltk.collocations import *

#naredimo seznam vseh besed v naši učni množici
all_words = [word for text, category in tokenizedlist for word in text]

#inicializacija razreda za iskanje kolokacij v našem seznamu besed
finder = BigramCollocationFinder.from_words(all_words)

#določimo filter - vsaka kolokacija se mora v množici pojaviti vsaj trikrat
finder.apply_freq_filter(3)

#inicializacija razreda z različnimi kriteriji za ocenjevanje kolokacij
bigram_measures = nltk.collocations.BigramAssocMeasures()

#iskanje in izdelava množice kolokacij, ki so bile najbolje ocenjene po
#kriteriju PMI. V množico dodamo le 100 najbolje ocenjenih kolokacij
collocationSet = set(finder.nbest(bigram_measures.pmi, 100))

#vključitev kolokacij v našo učno množico
for text, category in tokenizedlist:
     text.extend([bigram[0] + " " + bigram[1] for bigram in
                  nltk.bigrams(text) if bigram in collocationSet])
```

Po vključitvi kolokacij smo preizkusili še dve metodi za izboljšanje uspešnosti klasifikacije, ki sta precej specifični za našo učno množico in na drugačnih tekstih verjetno ne bi delovali. Ker predvsem tviti (to načeloma ne velja za komentarje) velikokrat vsebujejo različne URL naslove, ki po naši hipotezi ne vplivajo pozitivno na uspešnost klasifikacije, ampak predstavljajo le nepotreben šum v podatkih, smo v naši učni množici poiskali vse URL-je in jih preprosto nadomestili s frazo 'HTML' (preprosta izločitev URL-jev bi morda negativno vplivala na klasifikacijo, ker čeprav posamezni URL-ji zaradi svoje raznovrstnosti na klasifikacijo morda ne vplivajo pozitivno, pa samo vsebovanje URL-ja v tvitu morda klasifikatorju da določeno informacijo, ki je relevantna za uspešno klasifikacijo). Pri tem smo želeli uporabiti SpaCy-jevo metodo za iskanje URL naslovov v tekstu, vendar pa je ta metoda dosegljiva le znotraj cevovoda v sklopu z drugimi operacijami, zato je nismo uporabili, ampak smo napisali kar lastno funkcijo za iskanje URL naslovov (kode zaradi preprostosti same metode ne bomo dodali, je pa priložena v dodatku diplomske naloge).

Druga metoda za izboljšanje uspešnosti klasifikacije upošteva določene značilnosti spletne slovenščine, kjer se besede poudarja s pomočjo ponavljanja znakov. Če vzamemo za primer, v veliko tvitih lahko zasledimo podobne zapise kot so 'goooooool!!!!!', s tem pa želi pisec tvita poudariti lastno navdušenje nad zadetkom na nogometni tekmi. Zgoraj v slengu zapisana beseda gol je kot taka neprimerljiva z drugimi omembami gola v učni množici, zato smo s

pomočjo lastno napisane funkcije (tudi tu kode zaradi preprostosti ne bomo priložili, je pa priložena v dodatku diplomske naloge) poiskali vse žetone s ponavljajočimi znaki v besedilu in iz njih odstranili ponovitve. Iz zgornjega primera tako s pomočjo naše funkcije nastaneta žetona 'gol' in '!'.

S temi izboljšavami smo želeli izboljšati uspešnost našega najboljšega klasifikatorja, torej klasifikatorja s pomočjo naivnega Bayesa iz knjižnice Scikit-learn. V spodnji tabeli (Tabela 5.4) si poglejmo, če so zgoraj navedene operacije pripomogle k boljši uspešnosti klasifikacije.

	Točnost	**Preciznost**	**Priklic**	**F-mera**
Natančnost klasifikatorja brez izboljšav	0,670	0,686	0,643	0,664
Dodajanje kolokacij	0,672	0,687	0,644	0,665
Odstranitev URL-jev	0,671	0,687	0,644	0,665
Odstranitev ponavljanj	0,675	0,688	0,650	0,668
Vse izboljšave skupaj	0,677	0,686	0,653	0,669

Tabela 5.4: Meritve vpliva izboljšav na uspešnost klasifikatorja s pomočjo naivnega Bayesa iz knjižnice Scikit-learn

Če uvedemo vse izboljšave naenkrat, nam uspe točnost klasifikacije povečati za 0,7 odstotka ter F-mero za 0,5 odstotka. K izboljšanju največ pripomore odstranitev ponavljanj, ki kot samostojna izboljšava izboljša točnost za pol odstotka, F-mero pa za 0,3 odstotka. Najmanj k izboljšavi uspešnosti pripomore odstranitev URL-jev, kar je pričakovano, saj komentarji, ki predstavljajo polovico učne množice, URL-jev ponavadi ne vsebujejo.

Kot vidimo, nobena od zgoraj omenjenih izboljšav (filtriranje s pomočjo oblikoskladenjskega označevanja, filtriranje s pomočjo TF-IDF, vključitev kolokacij ter odstranitev URL-jev in ponavljanj) ni uspela korenito izboljšati uspešnosti klasifikacije. Problem verjetno predstavlja naša relativno majhna učna množica, ki preprosto ne vključuje dovolj označenih primerov za boljšo klasifikacijo. Že na začetku poglavja o klasifikaciji smo omenili, da pridobitev ali

izdelava učne množice pri strojnem učenju predstavlja problem predvsem za jezike z malo govorci. V slovenščini tako večja učna množica za analizo sentimenta ni javno dosegljiva (morda pa sploh še ne obstaja), ročna izdelava slednje pa predstavlja kar velik izziv. Zato bomo v nadaljevanju preverili še metodo za avtomatsko izdelavo učne množice. Pred tem pa bomo nekaj besed namenili še interoperabilnosti posameznih knjižnic.

5.8 Interoperabilnost knjižnic

Že v prejšnjih poglavjih (predvsem v poglavju o predprocesiranju in izboljšavah uspešnosti) smo lahko opazili, da lahko med seboj kombiniramo metode različnih knjižnic, v tem kratkem poglavju pa bomo predstavili nekaj splošnih ugotovitev, ki se tičejo sodelovanja med knjižnicami.

Opazimo lahko, da se določene knjižnice med seboj dobro dopolnjujejo. Tako smo v poglavju o predprocesiranju uporabili NLTK-jev tokenizator in njegov izhod v obliki seznama žetonov kot vhod dali knjižnici Lemmagen, ki je poskrbela za lematizacijo besed. Njen izhod smo nato uporabili za klasifikacijo v knjižnici Pattern ter NLTK. V poglavju o izboljšavah smo prav tako lahko opazili, da se da Patternovo metodo za filtriranje s pomočjo TF-IDF uporabiti tudi v sodelovanju s knjižnico NLTK. Po drugi strani smo NLTK-jevo metodo za iskanje kolokacij lahko uporabili tudi pri klasificiranju tvitov s pomočjo knjižnice Pattern.

Po drugi strani v tem poglavju nismo uporabljali metod knjižnic SpaCy, TextBlob (razen pri meritvi hitrosti klasifikacije) in PyNLPl. Knjižnice PyNLPl nismo mogli uporabiti zaradi pomanjkanja osnovnih metod za procesiranje naravnega jezika, knjižnico SpaCy pa nismo uporabili zaradi določene nepovezljivosti z drugimi knjižnicami. SpaCyjev cevovod morda pripomore k hitrosti izvajanja sklopa operacij znotraj cevovoda, vendar pa ima to določeno ceno. Zaradi cevovoda, ki kot argument sprejema le besedilni niz, bi bilo na primer treba za iskanje URL-jev znotraj učne množice (to funkcijo knjižnica SpaCy vsebuje) vse predhodne operacije obdelave teksta prav tako izvesti s pomočjo te knjižnice. Podobno velja za knjižnico TextBlog, kjer njen klasifikator kot vhod sprejema le besedilni niz, kar pomeni, da je spet potrebno vse prejšnje operacije predprocesiranja teksta izvesti s pomočjo te knjižnice. Edina metoda za procesiranje naravnega jezika iz teh dveh knjižnic, ki bi jo lahko pri našem klasificiranju teksta uporabili v sodelovanju z drugimi knjižnicami, je metoda za tokenizacijo, saj je ta metoda znotraj sekvence operacij za procesiranje naravnega jezika skoraj vedno prva.

Medtem ko nas prilagodljivost in interoperabilnost knjižnice NLTK ni presenetila, saj ta knjižnica ne vsebuje nobenih cevovodov in omogoča oziroma celo zahteva, da sami definiramo vsako še tako nepomembno pomožno funkcijo za procesiranje naravnega jezika,

pa nas je presenetila prilagodljivost knjižnice Pattern. Tako njeni klasifikatorji sprejemajo celo množico različnih vhodov (kot vhod lahko klasifikatorju damo niz, tokeniziran niz v obliki seznama, ali pa njegova notranja razreda 'Document' in 'Model'), nato pa klasifikator sam razbere, za kakšen vhod gre, in ravna temu primerno. Podobno velja za Patternovo funkcijo 'parse', ki smo jo uporabljali v prejšnjih poglavjih.

Tako velja, da gre v končni fazi za kompromis med preprostostjo uporabe in hitrostjo na eni strani ter prilagodljivostjo in interoperabilnostjo na drugi strani. Knjižnica NLTK je tako zapletena za uporabo in nič kaj hitra, a zelo prilagodljiva, knjižnici TextBlob (z izjemo njenega klasifikatorja, ki se je izkazal za najpočasnejšega) in SpaCy pa sta hitri in preprosti za uporabo, a nista interoperabilni. Edina, ki ji je uspelo vsaj delno preseči ta kompromis, je knjižnica Pattern, ki je kljub precejšnji prilagodljivosti in interoperabilnosti tudi preprosta za uporabo. Seveda pa to delno preseganje kompromisa zahteva določeno časovno kazen, tako da ta knjižnica ni med najhitrejšimi.

5.9 Avtomatska izdelava učne množice tvitov za določanje sentimenta

Pri določanju sentimenta s pomočjo strojnega učenja velik problem povzroča pridobitev primerne učne množice. To še posebej velja za manjše naravne jezike z malo govorci, kot je slovenščina. Ročna izdelava dovolj velike in primerne učne množice je zamudno in drago opravilo, zato so se pojavili pristopi za avtomatsko izdelavo učne množice za učenje sentimenta.

Tako je leta 2009 na univerzi Stanford nastala učna množica 1.600.000 (od tega 800.000 pozitivnih in 800.000 negativnih) tvitov, kjer so bili tviti označeni glede na prezenco pozitivnih in negativnih 'emoticonov'. V tej zbirki so tako 'emoticoni' tisti, ki ponazarjajo negativen in pozitiven sentiment. Ta pristop je prvi predlagal Read leta 2005 [5]. Za primer, če tvit vsebuje znak :), potem je označen kot pozitiven, če pa vsebuje znak :(, je označen za negativnega. Oznake seveda zaradi avtomatičnega označevanja niso popolnoma pravilne, vendar pa je to množico preprosto in poceni izdelati.

Odločili smo se, da s pomočjo Twitter API-ja [30] in Pythonove knjižnice za uporabo tega API-ja, knjižnice Tweepy [15], naberemo množico slovenskih tvitov, ki vsebujejo pozitivne in negativne 'emoticone', ter jo uporabimo kot učno množico. Ker želimo tvite kategorizirati v tri kategorije (tiste s pozitivnim sentimentom, tiste z negativnim sentimentom in nevtralne tvite), v učni množici pa imamo le dva razreda (pozitivne in negativne tvite), smo se odločili za predelavo klasifikatorja s pomočjo naivnega Bayesa iz knjižnice NLTK (ta klasifikator se

je obnesel skoraj enako dobro kot klasifikator s pomočjo naivnega Bayesa iz knjižnice Scikit-learn, hkrati pa je bila njegova predelava lažje izvedljiva kot predelava klasifikatorja iz knjižnice Scikit-learn), da bi omogočal takšno vrsto klasifikacije. Bayes klasificira na podlagi že zgoraj navedene formule (5.1), in sicer za vsako značilko algoritem izračuna verjetnost, da pripada določeni kategoriji. Prispevki značilk posameznega tvita ali komentarja se zmnožijo in tako dobimo verjetnosti, da tvit ali komentar pripada določenemu razredu. Nato algoritem le še izbere razred, pri katerem je verjetnost, da mu tvit ali komentar pripada, največja.

Mi smo za osnovo vzeli algoritem naivnega Bayesa iz knjižnice NLTK in ga predelali tako, da v primeru, ko verjetnost, da tvit pripada najverjetnejšemu razredu, ne presega določenega praga, algoritem ta tvit preprosto označi kot nevtralen. Povedano drugače, med pozitivni in negativni razred smo vnesli nevtralno cono, kamor algoritem uvrsti tvite, za katere ni dovolj gotovo, da spadajo v pozitivni ali negativni razred. Ta verjetnostni prag smo poiskali empirično s preizkušanjem na naši ročno narejeni učni množici tvitov in komentarjev, iz katere smo izločili vse nevtralne tvite. Verjetnost najverjetnejšega razreda je tako morala preseči prag 0.55, da je bil tvit klasificiran v pozitivni ali negativni razred. Pri tako določenem pragu je klasifikator dosegal najboljšo, 57,3% točnost ter 48,1% preciznost in 54,4% priklic. Glede na majhnost učne množice (brez nevtralnih tvitov je ta pri 10-kratnem prečnem preverjanju obsegala manj kot 2000 tvitov) se nam ta rezultat ni zdel tako slab, prav tako je točnost presegala točnost večinskega klasifikatorja za okoli 15 odstotnih točk.

Tvite smo nabirali dva tedna (v obdobju od 24.7.2015 do 7.8.2015) in v tem času nabrali 264819 tvitov. Tvite smo najprej želeli nabrati s pomočjo filtra za lokacijo v Twitter API-ju, ki smo mu kot argument dali geolokacijo Ljubljane in območje omejili s polmerom 50 kilometrov. Tako nabrani tviti so se izkazali za neprimerne, saj je bilo več kot tri četrtine tvitov v drugih jezikih (predvsem angleščini in hrvaščini). Ker se tudi Twitterjev filter za jezik ni izkazal za dobrega (filter ni znal ločiti med slovenskim, hrvaškim, srbskim in češkim jezikom, zato je bila le manjšina tako nabranih tvitov v slovenščini), smo se odločili za kombinacijo filtriranja tvitov s pomočjo filtra za slovenski jezik ter filtriranja s pomočjo najpogostejših slovenskih besed, ki se redko uporabljajo v drugih jezikih:

"kakor", "prav", "zelo", "bolj", "sicer", "proti", "vendar", "potem", "seveda", "veliko", "predvsem", "sem", "bodo", "kar","med","lahko","pri", "kot", "in", "so", "ki", "tudi"

Na ta način smo dobili množico večinoma slovenskih tvitov, ki je vsebovala 264919 tvitov. Ko smo iz te množice izločili tvite z 'emoticoni', smo dobili 6012 tvitov, ki vsebujejo pozitivni 'emoticon', ter le 721 tvitov, ki vsebujejo negativni 'emoticon'. Majhno število tvitov z negativnim 'emoticonom' nas je negativno presenetilo. Iz vseh tvitov smo nato izločili 'emoticone', saj nismo želeli, da bi ti preveč vplivali na klasifikacijo, jim pripisali oznake ter

jih shranili v že prej velikokrat omenjeni seznam dvojic (tvit, oznaka), ki je primeren za nadaljnjo obdelavo in izračun značilk.

Nato smo naš modificirani klasifikator naučili na naši avtomatsko narejeni učni množici ter njegovo uspešnost preizkusili na naši ročno narejeni množici tvitov in komentarjev, ki smo jo uporabljali že v prejšnjih poglavjih. Klasifikator je dosegel 44,8% točnost, 44,2% preciznost, 46,9% priklic in 45,5% F-mero. Čeprav točnost klasifikatorja za okoli 2% presega točnost večinskega klasifikatorja, so rezultati dokaj slabi. Na to verjetno vpliva predvsem majhnost učne množice (predvsem pomanjkanje negativnih tvitov), seveda pa ne smemo zanemariti tudi šuma v podatkih (vsi tviti s pozitivnimi 'emoticoni' niso pozitivni, prav tako pa vsi tviti z negativnimi 'emoticoni' niso negativni).

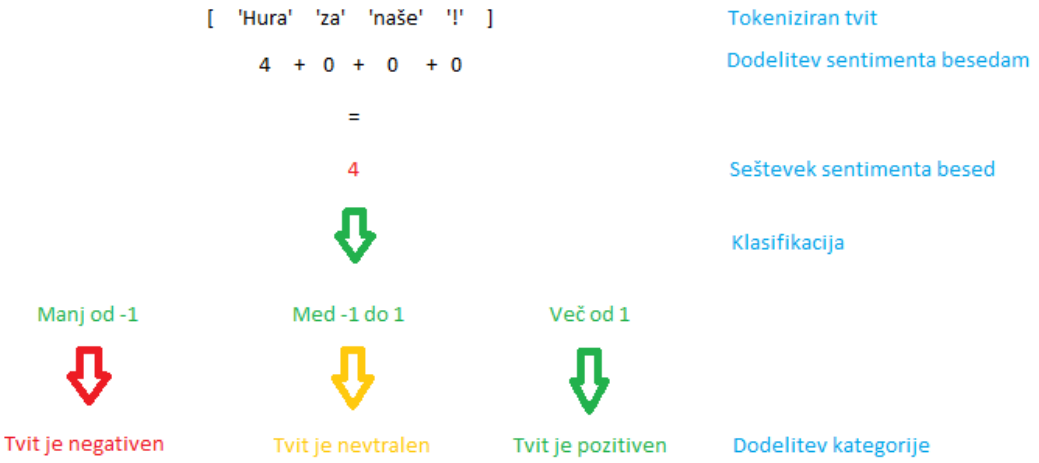
Tako lahko zaključimo, da izdelava avtomatske učne množice za slovenščino ni enostavna naloga, saj bi za zadostno število tvitov potrebovali zelo veliko časa (ob isti frekvenci tvitov z negativnimi 'emoticoni' bi za zbiranje okoli 10000 takšnih tvitov potrebovali okoli 7 mesecev). Očitno pri jezikih z malo govorci, kot je slovenščina, ni enostavne in dobre alternative za kvalitetno ročno narejeno učno množico.

5.10 Leksikalni pristop

V nadaljevanju smo se odločili na hitro preizkusiti tudi leksikalni pristop in poskušali tvite in komentarje klasificirati s pomočjo seznama čustveno zaznamovanih slovenskih besed, ki jim je bil pripisan pozitiven ali negativen sentiment. Večina večjih in bolj podrobnih seznamov takih besed je plačljiva, zato nam jih žal ni uspelo dobiti, prosto dostopen je bil le seznam v slovenščino prevedenih besed AFINN [23], ki vsebuje približno 3200 čustveno zaznamovanih besed s pripisanim sentimentom. Te besede imajo sentiment določen s pomočjo številk od -5 do +5, kjer beseda z oznako -5 pomeni čustveno zelo negativno zaznamovano besedo, oznaka +5 pa čustveno zelo pozitivno zaznamovano besedo. Za primer, beseda 'svinja' ima oznako -5, beseda 'hura' pa oznako +5. Manj čustveno zaznamovane besede so razporejene po intervalu med -5 in 5 glede na čustveno zaznamovanost.

5.10.1 Metoda merjenja in rezultati meritev

Za začetek smo vsak tvit in komentar procesirali na enak način kot pri strojnem učenju. Nato smo tvit klasificirali na podlagi postopka, ki je predstavljen s shemo (Slika 5.7).



Slika 5.7: Prikaz klasifikacije tvita ali komentarja s pomočjo leksikalne metode

Tvite in komentarje smo tokenizirali in lematizirali ter na ta način dobili sezname lematiziranih besed in ločil. Nato smo napisali funkcijo, ki se je sprehodila čez ta seznam besed in za vsako besedo pogledala, če je v seznamu čustveno zaznamovanih besed. V primeru, da je beseda bila v seznamu, smo sentimentu tvita in komentarja, ki je imel privzeto vrednost 0, prišteli sentiment te besede in tako na koncu za vsak tvit in komentar dobili njegovo skupno vrednost sentimenta. Tvite in komentarje smo na koncu klasificirali v tri razrede (pozitiven, negativen, nevtralen) na podlagi te skupne vrednosti sentimenta. Pri meritvah smo empirično določili tri različne nevtralne intervale za klasifikacijo, in sicer intervale (-2,2), (-1,1), (0,0). Če je vrednost sentimenta tvita padla v interval, smo ga klasificirali kot nevtralnega, če je padla pod interval, smo ga klasificirali kot negativnega, in če je padla nad interval, smo ga klasificirali kot pozitivnega. Nato smo izmerili točnost te klasifikacije na enak način kot pri strojnem učenju. Dobili smo naslednje rezultate (Tabela 5.5):

Interval	(-2,2)	(-1,1)	(0,0)
Točnost	0,396	0,411	0,402

Tabela 5.5: Rezultati meritev točnosti leksikalne metode

Kot vidimo, je naš klasifikator najboljšo točnost dosegel pri intervalu (-1,1), in sicer 41%, kar pa je še vedno slabše od večinskega klasifikatorja pri strojnem učenju, ki je dosegel 43% točnost. Očitno je, da takšna klasifikacija s pomočjo leksikalne metode ni zadovoljiva. Morda pri interpretaciji tega slabega rezultata pomaga dejstvo, da je v 50% primerov sentimentalna

vrednost tvita ali komentarja imela vrednost 0, iz česar lahko sklepamo, da polovica tvitov in komentarjev (če zanemarimo zelo redke primere, kjer je seštevek pozitivnih in negativnih besed znašal točno 0) ni vsebovala niti ene besede iz tabele čustveno zaznamovanih besed. Tako lahko zaključimo, da je tabela čustveno zaznamovanih besed očitno premalo obsežna ali pa vsebuje nerelevantne besede. Na ta problem smo opozarjali že na začetku poglavja, kjer smo opozorili na nefleksibilnost leksikalne metode, ki težko sledi spremembam v še kako spreminjajočem jeziku na spletu.

Poglavje 6 Sklepne ugotovitve

V diplomi smo se lotili procesiranja naravnega jezika v programskem jeziku Python. Temeljito smo preučili pet knjižnic za procesiranje naravnega jezika ter jih med seboj primerjali po funkcionalnosti, hitrosti in točnosti. Navedli in predstavili smo vse glavne funkcionalnosti knjižnic ter knjižnice med seboj primerjali po hitrosti tokenizacije ter hitrosti in točnosti lematizacije, oblikoskladenjskega označevanja in klasifikacije. Na koncu smo uporabili znanje, ki smo ga pridobili v prejšnjih poglavjih, in poskusili s pomočjo različnih metod in funkcij iz različnih knjižnic določiti sentiment množici slovenskih tvitov ter pri tem doseči čim boljšo točnost. V zaključnem komentarju pa bomo navedli še glavne ugotovitve, do katerih smo prišli med izdelavo te diplomske naloge.

6.1 Ocene knjižnic

Za začetek lahko rečemo, da knjižnica PyNLPl očitno ni primerna za večino nalog, ki pridejo v poštev pri procesiranju naravnega jezika, saj preprosto ne vsebuje nekaterih funkcionalnosti in metod, ki bi jih knjižnica za procesiranje naravnega jezika morala imeti. Vsebuje le metodo za tokenizacijo, pa še ta se je na hitrostnem testu izkazala za najpočasnejšo. Tako lahko rečemo, da knjižnice PyNLPl kljub nekaterim metodam za procesiranje naravnega jezika, ki jih knjižnica vsebuje, za kakršnokoli resno procesiranje naravnega jezika v Pythonu ne priporočamo.

Knjižnica TextBlob nas je s svojo hitrostjo navdušila pri tokenizaciji in oblikoskladenjskem označevanju, kjer se je izkazala za najhitrejšo, za precej hitro pa se je izkazala tudi pri lematizaciji. Pri lematizaciji in oblikoskladenjskem označevanju se je prav tako izkazala s svojo solidno točnostjo. Po drugi strani nas je razočarala pri klasifikaciji, saj se je njena implementacija klasifikatorja s pomočjo naivnega Bayesa iz knjižnice NLTK izkazala za najpočasnejšo. Knjižnica nas je prav tako navdušila s preprostostjo uporabe, ki pa ima na žalost za posledico manjšo prilagodljivost ter oteženo souporabo knjižnice v sodelovanju z drugimi knjižnicami.

Veliko stvari, ki smo jih omenili pri knjižnici TextBlob, velja tudi za knjižnico SpaCy. Knjižnica je zelo hitra in se je po hitrosti izkazala pri vseh nalogah, ki smo jih ji zastavili. Verjetno bi se po hitrosti še bolje izkazala na večjih besedilih od tistih, na katerih smo jo

preizkušali mi, saj je bilo pri vseh hitrostnih primerjavah razvidno, da čas izvajanja s številom besed ali znakov narašča počasneje, kot pri ostalih knjižnicah. Po drugi strani ta knjižnica potrebuje nekaj časa za začetno inicializacijo, zato je morda manj primerna za procesiranje manjših tekstov. Knjižnica SpaCy je tudi dokaj preprosta za uporabo, presenetila pa nas je tudi njena točnost pri oblikoskladenjskem označevanju. Pri tej knjižnici nas je po drugi strani najbolj motilo pomanjkanje raznih dodatnih funkcionalnosti in njena precejšnja nefleksibilnost. Večino njenih metod je dostopnih le v sklopu z drugimi metodami v njenem cevovodu, kar knjižnico naredi nerodno za uporabo skupaj z drugimi knjižnicami za procesiranje naravnega jezika.

Knjižnica Pattern je na nas naredila mešan vtis. Zadovoljni smo bili s širokim naborom funkcionalnosti ter njeno prilagodljivostjo, v večini primerov pa nismo bili najbolj zadovoljni z njeno hitrostjo in točnostjo. Pri tokenizaciji in oblikoskladenjskem označevanju se je izkazala za povprečno hitro, vendar pa se pri oblikoskladenjskem označevanju ni izkazala s točnostjo. Prav tako nas je motila počasnost njenega lematizatorja v funkciji 'parse', ki pa se je izkazal z dobro točnostjo. Tudi njeni algoritmi za klasifikacijo se niso izkazali s točnostjo. Po drugi strani so nam precej prav prišle določene funkcije, ki jih nismo našli v nobeni od ostalih knjižnic.

Zanimivo je bilo tudi delo s knjižnico NLTK. Knjižnica nima para, kar se tiče prilagodljivosti in fleksibilnosti, prav tako se z njo po širokem naboru funkcionalnosti ne more primerjati nobena od ostalih knjižnic. Všeč nam je bil tudi lahek dostop do cele množice različnih besedilnih korpusov, prav tako je ta knjižnica zaradi svoje prilagodljivosti edina, ki omogoča izvajanje oblikoskladenjskega označevanja na slovenskih tekstih. V tej knjižnici nam je bilo všeč tudi to, da obstaja več alternativ za vsako operacijo, tako da se lahko sami odločimo, kateri algoritem želimo uporabiti. Po drugi strani so preizkusi pokazali, da so nekateri od teh alternativnih algoritmov zastareli in v praksi neuporabni zaradi slabe točnosti in počasnosti. Na splošno ta knjižnica ne blesti po hitrosti (motila nas je predvsem počasnost privzetega oblikoskladenjskega označevalnika), prav tako pa ni preprosta za uporabo, saj je treba za večino operacij napisati več kode kot v drugih knjižnicah.

Pri določanju sentimenta slovenskim tvitom in komentarjem s pomočjo strojnega učenja se je najbolje obnesel klasifikator iz knjižnice Scikit-learn, ki se je izkazal za najhitrejšega in najbolj točnega. Dobro se je obnesel tudi klasifikator s pomočjo naivnega Bayesa iz knjižnice NLTK, ki je dosegel skoraj enako točnost kot najboljši klasifikator, medtem ko se se vsi Patternovi klasifikatorji izkazali za manj kvalitetne. Lahko pa rečemo, da se knjižnici NLTK in Pattern zaradi svoje fleksibilnosti lahko med seboj dobro dopolnjujeta in sta edini dve

knjižnici od obravnavanih, ki sta sposobni dobrega medsebojnega sodelovanja in interoperabilnosti.

Za trenutek se posvetimo še kriteriju primerjave, ki ga sicer v naši diplomski nalogi še nismo omenili, in sicer dokumentaciji knjižnic. Pohvalili bi dokumentacijo knjižnice NLTK, ki je daleč najobširnejša in najpodrobnejša od vseh, poleg dokumentacije pa obstaja še ogromno dodatne literature, ki nam je bila v veliko pomoč. Prav tako bi pohvalili dokumentacijo knjižnice TextBlob, ki je dobro napisana in zelo pregledna. SpaCyjeva dokumentacija se nam je zdela precej nepregledna in slabo napisana, vendar pa smo ravno ob koncu pisanja te diplomske naloge opazili, da je bila stran z dokumentacijo popolnoma prenovljena, zato težko podamo sodbo trenutnega stanja. Pomanjkljiva in nepregledna se nam je zdela tudi dokumentacija za knjižnico PyNLPl. Dokumentacija za knjižnico Pattern se nam je na prvi pogled zdela dobro napisana, podrobna in pregledna, vendar smo po temeljitem pregledu opazili, da manjkajo določena pojasnila, ki bi olajšala razumevanje delovanja posameznih funkcij in razredov znotraj knjižnice.

6.2 Izboljšave

Možne bi bile določene izboljšave diplomske naloge. Smiselno bi bilo najti še kakšno Pythonovo knjižnico za procesiranje naravnega jezika. Prav tako bi lahko Pythonove knjižnice za procesiranje naravnega jezika primerjali z javanskimi knjižnicami za procesiranje naravnega jezika še po kakšnem drugem kriteriju kot le po funkcionalnosti. Medtem ko smo dokaj podrobno raziskali in med seboj primerjali posamezne funkcionalnosti knjižnic, pa bi lahko nekaj več časa posvetili tudi primerjavi dokumentacij teh knjižnic.

Pri določanju sentimenta s pomočjo strojnega učenja bi si želeli doseči boljšo točnost, kar bi nam morda lahko uspelo z dodatnim procesiranjem tvitov in komentarjev. Še boljše pa bi bilo pridobiti malce večjo učno množico v slovenščini. Nezadovoljni smo bili tudi z rezultati leksikalne metode za določanje sentimenta, kjer bi nam prav tako prišla prav malce večja množica besed z določenim sentimentom. Kakorkoli, zaradi slengovsko obarvanega teksta v elektronskih besedilih bi se morda tudi večja množica besed izkazala za neuspešno.

Zanimivo bi bilo izvesti tudi nekaj poskusov določanja sentimenta s pomočjo malce večje avtomatsko narejene učne množice na podlagi 'emoticonov', za izgradnjo katere nam je zmanjkalo časa. Glede na dejstvo, da je naša avtomatsko narejena učna množica kljub svoji majhnosti dala boljše rezultate od večinskega klasifikatorja, lahko rečemo, da takšen pristop v praksi deluje, je pa potreben določenih izboljšav. Tako naši poskusi lahko služijo kot dobro

izhodišče za nadaljno raziskovanje na področju izdelave avtomatskih učnih množic za določanje sentimenta.

6.3 Priporočila

V končni fazi bi knjižnico SpaCy priporočili uporabnikom, ki želijo procesirati velike količine besedil v hitrem času, knjižnico TextBlob bi priporočili manj izkušenim uporabnikom, ki se ne spoznajo dobro na procesiranje naravnega jezika, knjižnico NLTK pa vsem, ki se želijo bolje spoznati s procesiranjem naravnega jezika ali pa želijo zelo prilagodljivo knjižnico, ogromno funkcionalnosti, možnost alternativ ter fleksibilnost. Knjižnica Pattern je zaradi preproste uporabe in velikega nabora funkcionalnosti dobra izbira predvsem za uporabnike, ki si želijo fleksibilnosti in preprostosti implementacije.

Zaključujemo z ugotovitvijo, da Python vsekakor je dober programski jezik za procesiranje naravnega jezika, saj ponuja množico dobrih knjižnic za procesiranje naravnega jezika, ogromno dokumentacije ter veliko podporno skupnost. Tako smo prepričani, da bo ta programski jezik še dolgo ostal med najpopularnejšimi jeziki za procesiranje naravnega jezika.

Literatura

[1] A good POS tagger in about 200 lines of Python. Dostopno na http://honnibal.wordpress.com/2013/09/11/a-good-part-of-speechpos-tagger-in-about-200-lines-of-python/ (julij 2015)

[2] Dependency trees, Natural Language Processing, and Google Ngrams. Dostopno na http://allthingslinguistic.com/post/51508922660/dependency-trees-natural-language-processing-and (avgust 2015)

[3] J. Perkins, *Python text processing with NLTK 2.0 Cookbook*, Packt Publishing, 2010.

[4] J. Perkins, *Python text processing with NLTK 3 Cookbook*, Packt Publishing, 2014.

[5] Jasmina Smailović, *Sentiment analysis in streams of microblogging posts*. Doktorska disertacija, Mednarodna podiplomska šola Jožefa Štefana, Ljubljana, 2014.

[6] Knjižnica ElementTree. Dostopno na https://docs.python.org/2/library/xml.etree.elementtree.html (avgust 2015)

[7] Knjižnica Gensim. Dostopno na http://radimrehurek.com/gensim/ (avgust 2015)

[8] Knjižnica NLTK. Dostopno na http://www.nltk.org (julij 2015)

[9] Knjižnica OpenNLP. Dostopno na http://opennlp.apache.org/ (avgust 2015)

[10] Knjižnica Pattern. Dostopno na http://www.clips.ua.ac.be/pattern (julij 2015)

[11] Knjižnica Py4J. Dostopno na http://www.py4j.org/ (avgust 2015)

[12] Knjižnica pyNLPl. Dostopno na http://pythonhosted.org/PyNLPl/ (julij 2015)

[13] Knjižnica SpaCy. Dostopno na http://honnibal.github.io/spaCy/ (julij 2015)

[14] Knjižnica TextBlob. Dostopno na http://textblob.readthedocs.org/en/dev/ (avgust 2015)

[15] Knjižnica Tweepy. Dostopno na http://docs.tweepy.org/en/v3.2.0/ (avgust 2015)

[16] Knjižnica Wordnet. Dostopno na: http://wordnet.princeton.edu/ (avgust 2015)

[17] Korpus Gutenberg. Dostopno na: http://www.gutenberg.org/wiki/Main_Page (avgust 2015)

[18] Korpus Treebank. Dostopno na: https://www.cis.upenn.edu/~treebank (avgust 2015)

[19] M. Juršič, I. Mozetič, N. Lavrač, Learning Ripple down rules for efficient lemmatization, v zborniku *Proceedings of the 10th International Multiconference Information Society 2007*, vol. A, str. 206-209, 2007

[20] Maks Horvat, *Orodja za tekstovno rudarjenje v slovenščini*, Diplomsko delo, Fakulteta za računalništvo in informatiko, Ljubljana, 2007.

[21] Matjaž Juršič, *Implementacija učinkovitega sistema za gradnjo, uporabo in evaluacijo lematizatorjev tipa RDR*. Diplomsko delo, Fakulteta za računalništvo in informatiko, Ljubljana, 2007.

[22] Projekt Sporazumevanje v slovenskem jeziku. Dostopno na: http://www.slovenscina.eu/ (julij 2015)

[23] Rok Martinc, *Merjenje sentimenta na družbenem omrežju Twitter: izdelava orodja ter evaluacija*. Magistrsko delo, Fakulteta za družbene vede, Ljubljana, 2013.

[24] S. Bird, E. Klein, E Loper, *Natural Language Processing with Python*, Sebastopol: O'Reilly media, 2009.

[25] Text classification for sentiment analysis – precision and recall. Dostopno na http://streamhacker.com/2010/05/17/text-classification-sentiment-analysis-precision-recall/ (julij 2015)

[26] Učna množica tvitov in komentarjev na temo športa in politike. Dostopno na http://nl.ijs.si/janes/interno/rezultati/#ucne-mnozice (april 2015)

[27] Učni korpus ssj500k. Dostopno na: http://www.slovenscina.eu/tehnologije/ucni-korpus (julij 2015)

[28] Wikipedia: Outline of natural language processing. Dostopno na http://en.wikipedia.org/wiki/Natural_Language_Toolkit (avgust 2015)

[29] Knjižnica Scikit-learn. Dostopno na http://scikit-learn.org/stable/ (avgust 2015)

[30] Twitter API. Dostopno na http://dev.twitter.com/overview/documentation (avgust 2015)

[31] John M. Zelle, *Python programming: An introduction to computer science*, Wilsonville : Franklin, Beedle, 2004.

[32] Projekt JOS: Jezikoslovno označevanje slovenskega jezika. Dostopno na http://nl.ijs.si/jos/ (avgust 2015)

[33] Knjižnica CoreNLP. Dostopno na http://nlp.stanford.edu/software/corenlp.shtml (avgust 2015)

[34] Knjižnica ClearNLP. Dostopno na http://github.com/clir/clearnlp (avgust 2015)

[35] Knjižnica Timeit. Dostopno na http://docs.python.org/2/library/timeit.html (avgust 2015)

[36] Wikipedia: Brown clustering. Dostopno na http://en.wikipedia.org/wiki/Brown_clustering (avgust 2015)

[37] M.F. Porter , An algorithm for suffix stripping, *Program*, št. 14, zv. 3, str. 130-137, 1980.

[38] Wikipedia: Brill tagger. Dostopno na http://en.wikipedia.org/wiki/Brill_tagger (avgust 2015)

[39] Knjižnica MontiLingua. Dostopno na http://web.media.mit.edu/~hugo (avgust 2015)

[40] T. Brants, TnT - a statistical part-of-speech tagger, *ANLP*, št. 6, str. 224-231, 2000.

[41] C.D. Paice, Another stemmer, *SIGIR Forum*, št. 24, zv. 3, str. 56-61, 1990.